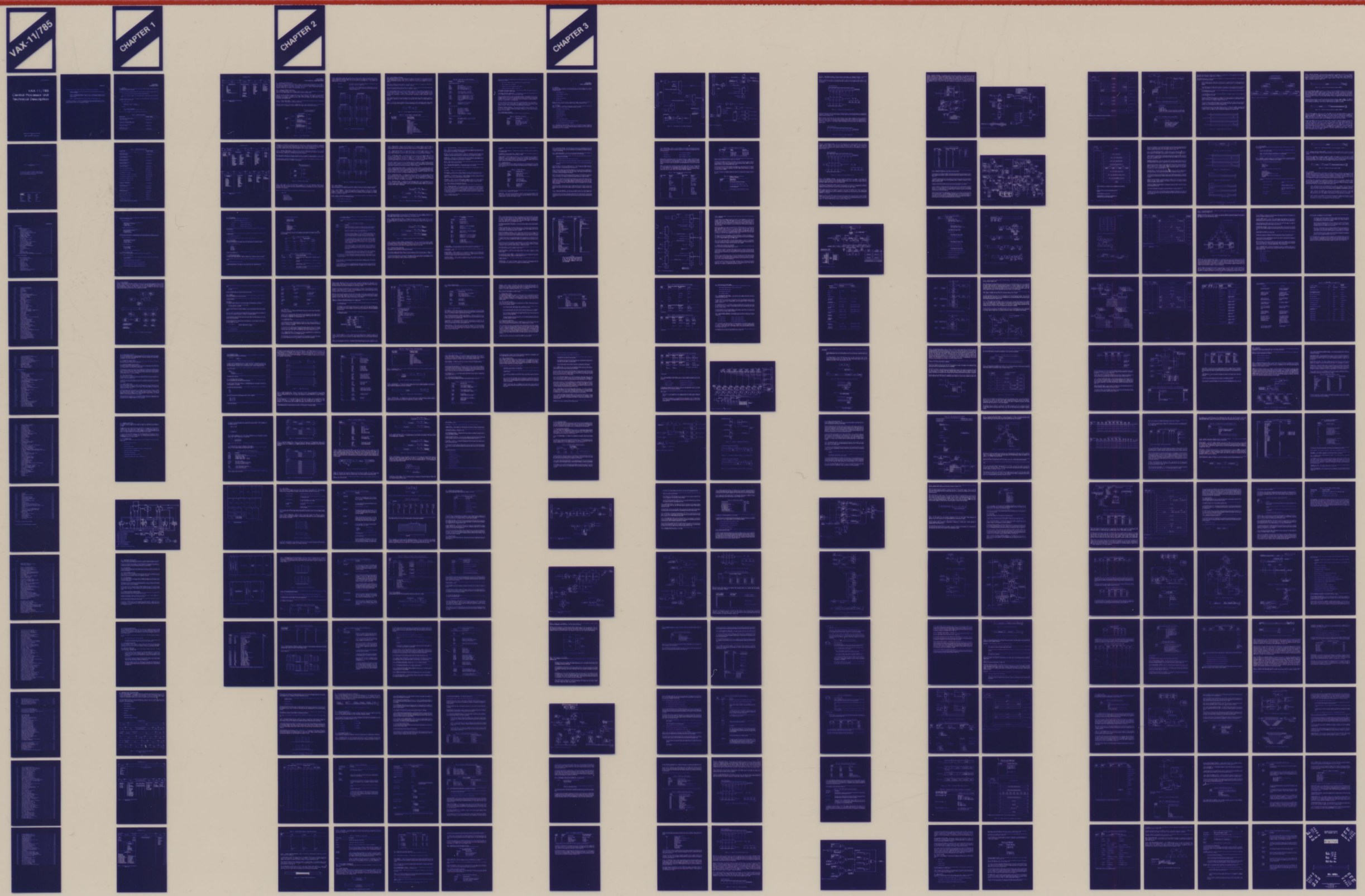
VAX-11/785

CENTRAL PROCESSOR UNIT
TECHNICAL DESCRIPTION

EP-KA785-TD-001
1 OF 2 JUN 1985
COPYRIGHT© 1985

digital
MADE IN USA

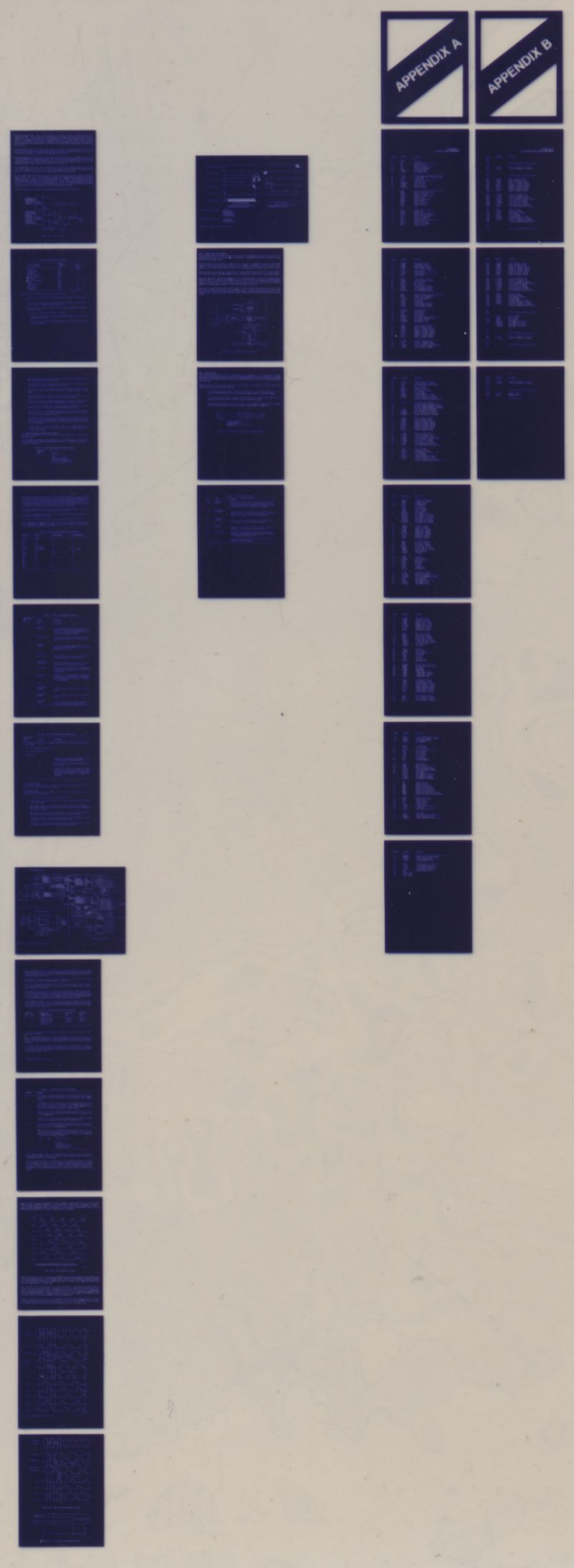


VAX-11/785

CENTRAL PROCESSOR UNIT
TECHNICAL DESCRIPTION

EP-KA785-TD-001
2 OF 2 JUN 1985
COPYRIGHT© 1985

digital
MADE IN USA



VAX-11/785

EK-KA785-TD-001

VAX-11/785 Central Processor Unit Technical Description

Prepared by Educational Services
of
Digital Equipment Corporation

First Edition, December 1984

Copyright © 1984 by Digital Equipment Corporation
All Rights Reserved

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document

Printed in U S A

The following are trademarks of Digital Equipment Corporation

digital™	DECtape	Rainbow
DATATRIEVE	DECUS	RSTS
DEC	DECwriter	RSX
DECmate	DIBOL	UNIBUS
DECnet	MASSBUS	VAX
DECset	PDP	VMS
DECsystem-10	P/OS	VT
DECSYSTEM-20	Professional	Work Processor

CONTENTS

1 INTRODUCTION

1.1 GENERAL 1-1
1.2 SYSTEM INTRODUCTION 1-3
1.2.1 Console Subsystem 1-4
1.2.2 Central Processor Unit (CPU) 1-5
1.2.3 Floating-Point Accelerator (FPA) 1-5
1.2.4 Synchronous Backplane Interconnect (SBI) 1-5
1.2.5 Main Memory Subsystem 1-5
1.2.6 UNIBUS Adapter (UBA) 1-5
1.2.7 MASSBUS Adapter (MBA) 1-6
1.3 CPU INTRODUCTION 1-6
1.3.1 Floating-Point Accelerator (FPA) 1-8
1.3.2 CPU Data Paths 1-8
1.3.3 Instruction Buffer and Decoder 1-8
1.3.4 CPU Microprocessor 1-8
1.3.5 Exceptions, Interrupts, and SBI Arbitration 1-8
1.3.6 System Clock (CLK) Generator 1-9
1.3.7 Console Interface Board (CIB) 1-9
1.3.8 TB, Cache, and SBI Control 1-9
1.4 READING THE CPU MICROCODE 1-10
1.4.1 Field Definition 1-15
1.4.2 Value Definitions 1-15
1.4.3 Microinstruction Definition 1-15
1.4.4 Comments 1-16
1.4.5 Macros 1-16
1.4.6 Microaddress Labels 1-17
1.4.7 Control Store Memory Allocation 1-17
1.5 SYSTEM LAYOUT AND CPU MODULE UTILIZATION 1-18

2 FUNCTIONAL DESCRIPTION

2.1 SYSTEM ARCHITECTURE 2-1
2.1.1 Virtual and Physical Addressing 2-1
2.1.1.1 Virtual Address Space 2-1
2.1.1.2 Physical Address Space 2-2
2.1.1.3 Data Types 2-2
2.1.2 Integer Data 2-4
2.1.3 Floating-Point Data 2-4
2.1.3.1 Single-Precision Data 2-4
2.1.3.2 Double-Precision Data 2-5
2.1.3.3 Extended-Precision Data 2-6
2.1.4 Data Strings 2-7
2.1.4.1 Character String Data 2-7
2.1.4.2 Leading Separate Numeric String Data 2-8
2.1.4.3 Trailing Numeric String Data 2-9

2.1.4.4 Packed Decimal String Data 2-10
2.1.4.5 Variable-Length Bit Field Data 2-12
2.1.4.6 Queue Data 2-13
2.1.5 System Registers 2-14
2.1.5.1 General Registers 2-14
2.1.5.2 Stacks 2-14
2.1.5.3 Processor Registers 2-16
2.1.5.4 Processor Status Longword (PSL) 2-18
2.2 NATIVE MODE INSTRUCTION FORMATS 2-22
2.2.1 Operation Code (Opcode) 2-22
2.2.2 Operand Specifier (Opspec) 2-22
2.2.2.1 Addressing Types 2-22
2.2.2.2 Access Types 2-24
2.3 NATIVE MODE ADDRESSING 2-24
2.3.1 Branch Addressing 2-24
2.3.2 General Register Addressing 2-25
2.3.2.1 Register Mode 2-26
2.3.2.2 Register Deferred Mode 2-26
2.3.2.3 Autoincrement Mode 2-26
2.3.2.4 Autoincrement Deferred Mode 2-26
2.3.2.5 Autodecrement Mode 2-26
2.3.2.6 Displacement Mode 2-26
2.3.2.7 Displacement Deferred Mode 2-27
2.3.2.8 Summary of General Register Addressing Mode 2-27
2.3.3 Program Counter Addressing 2-27
2.3.3.1 Immediate Mode 2-29
2.3.3.2 Absolute Mode 2-29
2.3.3.3 Relative Mode 2-29
2.3.3.4 Relative Deferred Mode 2-30
2.3.3.5 Literal Mode 2-30
2.3.3.6 Summary of Program Counter Addressing Modes 2-31
2.3.4 Index Addressing 2-32
2.3.4.1 Register Deferred Indexed Mode 2-33
2.3.4.2 Autoincrement Indexed Mode 2-33
2.3.4.3 Autodecrement Indexed Mode 2-33
2.3.4.4 Autoincrement Deferred Indexed Mode 2-33
2.3.4.5 Immediate Indexed Mode 2-33
2.3.4.6 Absolute Indexed Mode 2-33
2.4 NATIVE MODE INSTRUCTIONS 2-34
2.4.1 Logical, Stack, and Address Instructions 2-36
2.4.1.1 Logical Integer Instructions 2-36
2.4.1.2 Logical Bit Instructions 2-38
2.4.1.3 Stack and Address Instructions 2-38
2.4.2 Integer Arithmetic and Floating-Point Instructions 2-38
2.4.2.1 Arithmetic Instructions 2-38
2.4.2.2 Extended-Precision Instructions 2-38
2.4.3 Character String Instructions 2-40
2.4.3.1 Move and Compare Instructions 2-40
2.4.3.2 Locate, Skip, and Match Instructions 2-41
2.4.3.3 Scan, Span, and CRC Instructions 2-41
2.4.4 Packed Decimal String Instructions 2-41
2.4.5 Variable-Length Bit Field Instructions 2-43
2.4.6 Queue Instructions 2-43

2.4.7 Branch and Jump Instructions 2-44
2.4.7.1 Unconditional Branch and Jump Instructions 2-44
2.4.7.2 Branch-on-Bit Instructions 2-46
2.4.7.3 Branch-on-Condition Instructions 2-46
2.4.7.4 Loop Control and Case Instructions 2-47
2.4.8 Branch/Return and Jump Subroutine Instructions 2-48
2.4.9 Procedure Call and Return Instructions 2-48
2.4.10 Special Purpose Instructions 2-49
2.4.11 Privileged Instructions 2-50
2.5 COMPATIBILITY MODE 2-52
2.5.1 PDP-11 Instruction Execution 2-52
2.5.2 Operating System Restrictions 2-52
2.5.3 PDP-11 Addressing Modes 2-53

3 LOGIC DESCRIPTION

3.1 GENERAL 3-1
3.1.1 CPU Busses 3-1
3.1.1.1 Microprogram Counter (UPC) Bus 3-1
3.1.1.2 Control Store (CS) Bus 3-2
3.1.1.3 Internal Data (ID) Bus 3-2
3.1.1.4 Memory Data (MD) Bus 3-2
3.1.1.5 Visibility (V) Bus 3-5
3.1.1.6 LSI-11 Data/Address (Q) Bus 3-5
3.1.1.7 Virtual Address (VA) Bus and Physical Address (PA) Bus 3-5
3.1.1.8 Synchronous Backplane Interconnect (SBI) 3-5
3.2 CPU MICROPROCESSOR 3-6
3.2.1 Joint Control Store (JCS) Module 3-6
3.2.1.1 JCS Address Jumper 3-6
3.2.1.2 Control Store Microword and Parity 3-6
3.2.1.3 Control Store Addressing 3-9
3.2.2 Microsequencer (USC) Module 3-9
3.2.2.1 Microsequencer Mode Control (Picosequencer) 3-11
3.2.2.2 Branch Decoder 3-14
3.2.2.3 Branch Multiplexer Enable Gating 3-17
3.2.2.4 Initialize (INIT) Mode 3-19
3.2.2.5 Normal Mode 3-19
3.2.2.6 Cache Stall (STALL) Mode 3-19
3.2.2.7 Microstack Operation 3-19
3.2.2.7.1 Subroutine Control 3-19
3.2.2.7.2 Microstack Addressing 3-22
3.2.2.7.3 Microstack Push 3-22
3.2.2.7.4 Microstack Pop 3-22
3.2.2.8 Microtrap (UTRAP) Mode 3-23
3.2.2.9 WCS Addressing 3-24
3.2.2.10 Maintenance Mode 3-27
3.3 INSTRUCTION BUFFER AND DECODER 3-27
3.3.1 Instruction Data Path (IDP) Module 3-28
3.3.1.1 Instruction Buffer (IB) Register 3-28
3.3.1.2 Instruction Buffer Addressing 3-28
3.3.1.3 Program Counter (PC) Updates 3-31
3.3.1.4 Memory Data (MD) Byte Shifter 3-31

3.3.1.5 IB Shift Multiplexer (SHF MUX) ... 3-33
3.3.1.6 Loading the Instruction Buffer ... 3-33
3.3.1.6.1 First Memory Fetch ... 3-35
3.3.1.6.2 Second Memory Fetch ... 3-35
3.3.1.6.3 Third Memory Fetch ... 3-37
3.3.1.6.4 Fourth Memory Fetch ... 3-38
3.3.1.7 Data Multiplexer (DMX) ... 3-38
3.3.2 Instruction Decoder (IRC) Module ... 3-42
3.3.2.1 Execution Point Counter ... 3-42
3.3.2.2 VAX Execution ROMs and Control Word ... 3-42
3.3.2.3 Mode Multiplexer (Mode Mux) ... 3-47
3.3.2.4 Specifier Decoding ... 3-49
3.3.2.5 Instruction Optimization ... 3-50
3.3.2.6 Branch Decode Logic ... 3-53
3.3.2.7 Size Select ... 3-53
3.3.2.8 Index (I) Mode Flag ... 3-55
3.3.2.9 Specifier Constants ... 3-55
3.3.2.10 Microsequencer Branch Conditions ... 3-57
3.3.2.11 Compatibility Mode Decoding ... 3-57
3.4 CPU DATA PATHS ... 3-60
3.4.1 Address Section ... 3-60
3.4.1.1 Virtual Instruction Buffer Address (VIBA) Register ... 3-64
3.4.1.2 Virtual Address (VA) Register ... 3-64
3.4.1.3 Virtual Address Multiplexer (VAMX) ... 3-65
3.4.1.4 Program Counter (PC) Register ... 3-66
3.4.1.5 PC Adder (PCADD) annd Number Multiplexer (NMX) ... 3-67
3.4.1.6 PC Multiplexer (PCMX) and PC Adder Multiplexer (PCAMX) ... 3-67
3.4.2 Data Section ... 3-67
3.4.2.1 Data Section Interface ... 3-67
3.4.2.2 Q and D Registers and Data Aligner (DAL) ... 3-69
3.4.2.3 Memory Data (MD) Bus Interface ... 3-77
3.4.3 Arithmetic Section ... 3-82
3.4.3.1 Arithmetic/Logic Unit (ALU) ... 3-82
3.4.3.2 ALU A Input Multiplexer (AMX) ... 3-82
3.4.3.3 ALU B Input Multiplexer (BMX) ... 3-86
3.4.3.4 ALU Shifter (SHF) ... 3-90
3.4.3.5 Constant Multiplexer (KMX) ... 3-93
3.4.3.6 Bit Mask (MASK) Generator ... 3-96
3.4.3.7 Scratchpad Register Sets ... 3-99
3.4.3.8 Register Log Stack (RLOG) and PC Save (PCSV) Register ... 3-102
3.4.4 Exponent Section ... 3-103
3.4.4.1 Exponent Arithmetic Logic Unit (EALU) ... 3-103
3.4.4.2 EALU A Input Multiplexer (EAMX) ... 3-106
3.4.4.3 EALU B Input Multiplexer (EBMX) ... 3-106
3.4.4.4 Floating Exponent (FE) Register ... 3-107
3.4.4.5 State Register ... 3-107
3.4.4.6 Shift Count Multiplexer (SMX) ... 3-107
3.4.4.7 Shift Count (SC) Register ... 3-107
3.5 EXCEPTIONS AND INTERRUPTS ... 3-108
3.5.1 General Description ... 3-110
3.5.1.1 Exceptions ... 3-110
3.5.1.2 Interrupts ... 3-111
3.5.1.3 Compatibility Mode ... 3-111

3.5.1.4 Differences 3-112
3.5.2 Interrupts 3-113
3.5.2.1 Interrupt Priority Levels (IPLs) 3-113
3.5.2.2 Vectors 3-115
3.5.2.3 Internal Hardware Interrupts 3-115
3.5.2.4 External Hardware Interrupts 3-115
3.5.2.5 Hardware Interrupt Register (HIR) 3-117
3.5.2.6 Software Interrupts 3-121
3.5.2.7 Software Interrupt Register (SIR) 3-122
3.5.3 Exceptions 3-122
3.5.3.1 Exception Vectors 3-123
3.5.3.2 Serious System Failure Exceptions (Aborts) 3-125
3.5.3.3 Exceptions During an Operand Reference (Faults) 3-126
3,5.3.4 Exceptions as the Result of an Instruction (Faults or Traps) 3-128
3.5.3.5 Arithmetic Traps 3-131
3.5.3.6 Microtraps 3-132
3.5.4 Microword Control of Interrupts and Exceptions 3-134
3.5.4.1 Interrupt and Exception Control (UIEK) Field 3-134
3.5.4.2 Miscellaneous (UMSC) Field 3-135
3.6 SYSTEM CLOCKS 3-137
3.6.1 Processor Clock 3-137
3.6.1.1 Clock Phase and Error Detection 3-139
3.6.1.2 Frequency Selection 3-139
3.6.1.3 Start/Stop/Step Control 3-139
3.6.1.4 Processor Timing 3-140
3.6.2 Time-of-Day Clock (TODC) 3-145
3.6.3 Interval Timer 3-146

APPENDIX A INSTRUCTION OPCODES

APPENDIX B EXTENDED-PRECISION OPCODES

TABLES

1-1	Related Documentation	1-1
1-2	Typical VAX-11/785 Hardware Layout	1-18
1-3	KA785 Module Utilization	1-21
2-1	Summary of Integer Data Ranges	2-3
2-2	Summary of Floating-Point Data Precision	2-3
2-3	Summary of Data String Precision	2-4
2-4	Leading Separate Numeric Sign Characters	2-8
2-5	ASCII Numeric Characters	2-9
2-6	Trailing Numeric Digit and Sign Representations	2-11
2-7	Packed Decimal Digit and Sign Representations	2-12
2-8	General Register Description	2-15
2-9	Processor Registers	2-17
2-10	Processor Status Longword (PSL) Description	2-19
2-11	Summary of Addressing Types	2-23
2-12	Summary of Access Types	2-24
2-13	General Register Addressing Modes	2-25
2-14	Summary of General Register Addressing Modes	2-28
2-15	Program Counter Addressing Modes	2-29
2-16	Floating Literals	2-31
2-17	Summary of Program Counter Addressing Modes	2-32
2-18	Summary of Index Addressing Modes	2-35
2-19	Summary of Data Terms	2-36
2-20	Logical, Stack, and Address Instructions	2-37
2-21	Integer Arithmatic and Floating-Point Instructions	2-39
2-22	Character String Instructions	2-40
2-23	Packed Decimal String Instructions	2-41
2-24	Variable-Length Bit Field Instructions	2-43
2-25	Queue Instructions	2-44
2-26	Branch and Jump Instructions	2-45
2-27	Branch-on-Condition Instructions (Signed)	2-46
2-28	Branch-on-Condition Instructions (Unsigned)	2-47
2-29	Branch-on-Condition Instructions (Carry or Overflow)	2-47
2-30	Branch/Return and Jump Subroutine Instructions	2-48
2-31	Procedure Call and Return Instructions.	2-48
2-32	Special Purpose Instructions	2-49
2-33	Privileged Instructions	2-50
3-1	ID Bus Register Address Assignments	3-3
3-2	UCID Field Control of the ID Bus	3-4
3-3	JCS Module Address Configuration Jumpers	3-6
3-4	UPC Multiplexer Input Selection	3-12
3-5	UBEN Field Control of the Microcode Branch Condition Sets	3-14
3-6	BR MUX (F:0) Input Selection for BUS UPC (02:00)	3-16
3-7	BR MUX (1F:10) Input Selection for BUS UPC (4:0)	3-16
3-8	USUB Field Control Functions	3-20
3-9	PROM Vector Address Ranges	3-23
3-10	Conditions Causing Machine Microtraps	3-23
3-11	WCS Address Selection on a WCS Load	3-26
3-12	ID Register Selection on a Power-Up or Maintenance Operation	3-26
3-13	Program Counter Updates.	3-31

3-14 IBA Control of the MD Byte Shifter 3-32
3-15 UIBC Field Control of the MD Byte Shifter 3-34
3-16 Address Mode Control of DMX Data 3-40
3-17 Mode Field Control of the Mode Multiplexer 3-45
3-18 Access Field Definitions 3-46
3-19 Operand Length and Type Bit Field Definitions 3-46
3-20 Operand Length and Type Definitions. 3-47
3-21 Native Mode Control of the Mode Multiplexer 3-47
3-22 Specifier Decode Outputs.. 3-50
3-23 Abort Address Conditions 3-51
3-24 Size Selection for PC Updates 3-54
3-25 UBEN Field Control of the ICL Branch Multiplexer 3-58
3-26 Execution Point Counter Generation of CM EXEC 3-59
3-27 UVAK Bit Control of the VA Register..... 3-65
3-28 UPCK Field Control of the PC 3-67
3-29 USI Field Control of the Q Register Shift Input 3-71
3-30 USI Field Control of the D Register Shift Input 3-72
3-31 DAL Level 1 Outputs to DAL Level 2 3-77
3-32 DAL Level 2 Outputs to Dal 3-78
3-33 DAL Outputs to DMX..... 3-78
3-34 VA (01:00) Control of the MDAL 3-79
3-35 VA (01:00) Control of D Register Bytes..... 3-80
3-36 VA (01:00) Control of the BAL 3-80
3-37 VA (01:00) Control of D Register Byte Parity 3-81
3-38 VA (01:00) Control of the Byte Mask 3-81
3-39 UDT Field Context Control 3-82
3-40 UALU Field ALU Select Functions..... 3-83
3-41 ALU Functions in Instruction Dependent Mode 3-84
3-42 UAMX Field Control of the AMX 3-85
3-43 UDT Field Control of the RAMX 3-85
3-44 USGN Field Control of the Source and Destination Sign 3-89
3-45 UBMX Field Control of the BMX 3-90
3-46 USI Field Control of SHF Input..... 3-92
3-47 USHF Field Control of SHF Output..... 3-93
3-48 UDT Field Control of SHF Output (USHF is Equal to 3)..... 3-93
3-49 UKMX Field Constant Control 3-95
3-50 USPO Field Selection of the Scratchpad Registers..... 3-100
3-51 Scratchpad Address Code Number (ACN)..... 3-101
3-52 UD Field Control of RA/RB Register Writes 3-102
3-53 UEALU Field Control of the EALU 3-105
3-54 UEBMX Field Control of the EBMX 3-106
3-55 USMX Field Control of the SMX..... 3-107
3-56 Shift Count (SC) Register Functions 3-108
3-57 Interrupt Conditions, IPL and Vector Assignments 3-114
3-58 Event Control of Vector Address (01:00)..... 3-115
3-59 Hardware Interrupt Register (HIR) Bit Usage 3-119
3-60 Exception Conditions and Assigned Vectors..... 3-124
3-61 Machine Check Parameter Locations 3-125
3-62 Machine Check Identification in Summary Parameter Byte 0..... 3-126
3-63 Reserved Address and Mode Exceptions 3-127
3-64 Compatibility Mode Exceptions 3-129
3-65 Arithmetic Trap Conditions 3-131

3-66 Microtrap Service Priorities 3-133
3-67 UIEK Field Interrupt and Exception Functions 3-134
3-68 IACK and EACK Functions of the UIEK Field 3-135
3-69 UMSC Field Miscellaneous Functions 3-136
3-70 MCR Control of System Clock Frequency 3-139
3-71 MCR Control of CPU Clock Operation 3-140
3-72 ICCS Register Bit Functions 3-147

FIGURES

1-1 VAX-11/785 System Block Diagram 1-4
1-2 Central Processor Unit (CPU) Block Diagram 1-7
1-3 Microword Format and Field Definitions 1-10
1-4 VAX-11/785 System Cabinet Layout, Typical Front View 1-19
1-5 CPU Backplane Logic Distribution 1-20

2-1 Virtual Address Space 2-1
2-2 Physical Address Space 2-2
2-3 Integer Data Formats 2-5
2-4 Floating/Double-Floating Data Formats 2-6
2-5 Extended-Precision Floating Data Formats 2-6
2-6 Character String Format 2-7
2-7 Leading Separate Numeric String Format 2-8
2-8 Trailing Numeric String Formats 2-9
2-9 Packed Decimal String Format 2-10
2-10 Variable Length Bit Field Format 2-12
2-11 Absolute Queue Entry Formats 2-13
2-12 Self-Relative Queue Entry Formats 2-14
2-13 Stack and Process Relationships 2-16
2-14 Processor Status Longword (PSL) 2-18
2-15 VAX-11 Native Mode Instruction Format 2-22
2-16 Branch Addressing Operand Qualifier 2-24
2-17 General Register Mode Operand Specifier 2-25
2-18 Displacement Mode Operand Specifier 2-26
2-19 Displacement Deferred Mode Operand Specifier 2-27
2-20 Immediate Mode Operand Specifier 2-29
2-21 Absolute Mode Operand Specifier 2-29
2-22 Relative Mode Operand Specifier 2-30
2-23 Relative Deferred Mode Operand Specifier 2-30
2-24 Literal Mode Operand Specifier 2-30
2-25 Floating Literal Format 2-31
2-26 Literal Fields in Floating/Double-Floating Operands 2-31
2-27 Index Mode Operand Specifier 2-32

3-1 Joint Control Store (JCS) Module Block Diagram 3-7
3-2 Control Store (CS) Address Space 3-9
3-3 Microsequencer (USC) Module Block Diagram 3-10
3-4 Picosequencer and Mode Multiplexers 3-13
3-5 Branch Decode Multiplexers 3-15
3-6 Branch Multiplexer Enable Gating 3-18
3-7 The Microstack (USTACK) 3-21
3-8 WCS Addressing Logic 3-25
3-9 Instruction Data Path (IDP) Module Block Diagram 3-29

3-10 Register Addressing Multiplexers 3-30
3-11 Instruction Register Fields in VAX Instructions 3-30
3-12 Instruction Register Fields in PDP-11 Instructions 3-30
3-13 Example of a Memory Data Byte Shift 3-32
3-14 Result of the First Memory Fetch 3-35
3-15 Result of the Second Memory Fetch 3-36
3-16 Result of a 1-Byte Right Shift 3-36
3-17 Result of the Third Memory Fetch 3-37
3-18 Result of a 2-Byte Right Shift 3-37
3-19 Result of the Fourth Memory Fetch 3-38
3-20 Data Multiplexer (DMX) Logic 3-39
3-21 Floating Point Short Literal Format 3-41
3-22 DMX Format of a Floating Point Short Literal 3-41
3-23 PDP-11 Branch Instruction Format in the Buffer Register 3-41
3-24 PDP-11 Instruction Format in the Buffer Register 3-41
3-25 Instruction Decode Logic 3-43
3-26 Execution Point Counter 3-44
3-27 VAX Execute ROMs and Control Word Format 3-44
3-28 Mode Multiplexer Selection 3-48
3-29 Specifier Decode Logic 3-49
3-30 Double Operand Optimizing Flowchart 3-52
3-31 Single Operand Optimizing Flowchart 3-52
3-32 Branch Decode Logic 3-53
3-33 Size Select Logic 3-54
3-34 Index (I) Mode Operand Specifiers 3-55
3-35 Index (I) Mode Flag Logic 3-56
3-36 Specifier Constants Logic 3-56
3-37 Microbranch Condition Multiplexers 3-58
3-38 PDP-11 Instruction Buffer Register Format 3-59
3-39 Compatibility Mode Decode Logic 3-61
3-40 CPU Data Path Block Diagram 3-62
3-41 Microword Fields Used for Data Path Control 3-63
3-42 VIBA and VA Registers and Multiplexers 3-64
3-43 VAMX Data Formats 3-64
3-44 Program Counter (PC) Register and Multiplexers 3-66
3-45 Data/Arithmetic Section Interface 3-68
3-46 Packed Floating Point Format 3-69
3-47 Unpacked Floating Point Format 3-69
3-48 UQK Field Control of the Q Register 3-70
3-49 UQK Field Control of QMX Input Selection 3-71
3-50 UDK Field Control of the D Register 3-73
3-51 UDK Field Control of DMX Input Selection 3-74
3-52 Decimal String Memory Format 3-75
3-53 Swapped Decimal Number Format 3-75
3-54 Data Aligner (DAL) Shift Formats 3-75
3-55 Data Aligner Logic 3-76
3-56 Memory Data (MD) Bus Interface 3-77
3-57 ALU A Input Multiplexer (AMX) 3-85
3-58 A Input Multiplexer (AMX) Control 3-87
3-59 ALU B Input Multiplexer (BMX) 3-89
3-60 BMX Data Formats on the ALU B Inputs 3-91
3-61 ALU Shifter (SHF) Logic 3-92
3-62 ALU Shifter (SHF) Data Formats 3-94

3-63 Constant Multiplexer (KMX) ... 3-95
3-64 Bit Mask (MASK) Generator Logic ... 3-96
3-65 Example of Bit Pattern Generation ... 3-97
3-66 Bit Field to be Extracted ... 3-98
3-67 Generating Bit Pattern A ... 3-98
3-68 Resulting Bit Pattern B ... 3-98
3-69 Resulting Bit Pattern of the Extract Field ... 3-98
3-70 Scratchpad Register Logic ... 3-99
3-71 RLOG Entry Format ... 3-102
3-72 Exponent Section Logic ... 3-104
3-73 BMX Data in Packed Floating-Point Format ... 3-104
3-74 Exception and Interrupt Structure ... 3-109
3-75 Interrupt Request Arbitration Logic ... 3-113
3-76 Forming the Interrupt Vector ... 3-116
3-77 Generating Interrupt Vector Bits ⟨08:02⟩ ... 3-116
3-78 Interrupt Summary Read and Response Formats ... 3-117
3-79 Generating Vector Register Bits ⟨25:16⟩ ... 3-118
3-80 Vector Register ... 3-118
3-81 Hardware Interrupt Register (HIR) ... 3-118
3-82 Software Interrupt Request Register (SIRR) ... 3-121
3-83 Software Interrupt Summary Register (SISR) ... 3-121
3-84 Asynchronous System Trap Level Register (ASTR) ... 3-122
3-85 Software Interrupt Register (SIR) ... 3-122
3-86 Microtrap Logic ... 3-132
3-87 Clock (CLK) Module Block Diagram ... 3-138
3-88 CPU and SBI Time States ... 3-141
3-89 CPU Time State Generator Timing ... 3-142
3-90 SBI Time State Generator Timing ... 3-143
3-91 CPU Power Up Sequencer Timing ... 3-143
3-92 CPU Power Fail Sequencer Timing ... 3-144
3-93 Time-of-Day Clock (TODC) ... 3-145
3-94 Interval Clock Control/Status (ICCS) Register ... 3-146

PREFACE

This manual consists of three chapters that provide the following levels of information

- Chapter 1, **Introduction** provides a general introduction to the system and the Central Processor Unit (CPU) hardware

- Chapter 2, **Functional Description** introduces the VAX-11 architecture and describes the CPU registers It also describes the VAX-11 address and data types and briefly explains the native mode instructions

- Chapter 3, **Logic Description** describes the CPU logic to the block diagram level

The manual is intended to be a resource for training, field service, and manufacturing, and for use as a general reference document It is one of many documents that support the VAX-11/785 system Prior training or experience with VAX-11 architecture is assumed.

CHAPTER 1

CHAPTER 1
INTRODUCTION

1.1 GENERAL

This chapter introduces the VAX-11/785 system elements and defines the logic contained in the KA785 Central Processor Unit (CPU) backplane.

This manual is one of a set of three technical documents that support the CPU The other manuals are:

VAX-11/785 Console Interface Technical Description (EK-KC785-TD)

VAX-11/785 Translation Buffer, Cache and SBI Control Technical Description (EK-MM785-TD)

For further information on the VAX-11 instruction set and addressing modes, refer to the following DIGITAL courseware and documents.

Audio-visual courses:

Introduction to VAX-11 - Concepts
Introduction to VAX-11 - Instruction Set

Handbooks:

VAX Architecture Handbook
Computer Programming and Architecture - The VAX-11

Table 1-1 provides a complete list of the related hardware documentation.

Table 1-1 Related Documentation

Document Title	Document Number
VAX-11/785 Central Processor Unit Technical Description	EK-KA785-TD
VAX-11/785 Translation Buffer, Cache and SBI Control Technical Description	EK-MM785-TD
VAX-11/785 Console Interface Technical Description	EK-KC785-TD
MS780E Memory System Technical Description	EK-MS78E-TD

Table 1-1 Related Documentation (Cont)

Document Title	Document Number
MS780 Memory System Technical Description	EK-MS780-TD
FP785 Floating Point Accelerator Technical Description	EK-FP785-TD
DW780 UNIBUS Adapter Technical Description	EK-DW780-TD
RH780 MASSBUS Adapter Technical Description	EK-RH780-TD
VAX-11/780 Power System Technical Description	EK-PS780-TD
VAX-11/785 Installation Manual	EK-SI785-IN
VAX-11/785 Hardware User's Guide	EK-11785-UG
VAX Diagnostic User Guide	EK-VX11D-UG
VAX Maintenance Handbook, VAX Systems	EK-VAXV1-HB
VAX Maintenance Handbook, VAX-11/780	EK-VAXV2-HB
VAX Hardware Handbook, 1982-83	EB-21710-20
VAX Architecture Handbook, 1981	EB-19580-20
Computer Programming and Architecture - The VAX-11	EY-AX008-DP
CI780 Computer Interconnect Technical Description	EK-CI780-TD
DS780 Diagnostic System User Guide	EK-DS780-UG
DS780 Diagnostic System Technical Description	EK-DS780-TD
System Maintenance Print Set	MP01747-*
Console Maintenance Print Set	MP01748-*
CPU Maintenance Print Set	MP01749-*
FPA Maintenance Print Set	MP01754-*

* The latest revision is sent if none is specified.

Hardcopy documents may be ordered through the nearest Digital Equipment Corporation sales office or from one of the following sources

Hardware documents:

Publishing and Circulation Services, NRO3-1/W3
Digital Equipment Corporation
10 Forbes Road
Northboro, MA 01532

Handbooks and software manuals

Software Distribution Center, NRO2-1/J6
Digital Equipment Corporation
444 Whitney Street
Northboro, MA 01532

Technical and service documents, including maintenance print sets and diagnostic listings, are also available on microfiche Information on microfiche libraries can be obtained from one of the following sources

DIGITAL Field Service

Micropublishing Systems, FPO/B5
Digital Equipment Corporation
30 North Avenue
Burlington, MA 01803

Customers and OEMs

Digital Equipment Corporation
P O Box CS2008
Nasuha, NH 03061

(In the U S , you may call 1-800-258-1710)

1.2 SYSTEM INTRODUCTION

Figure 1-1 is a basic block diagram introducing the following main system elements

- LSI-11 console subsystem
- KA785 central processor unit (CPU) which may include the optional FP785 floating-point accelerator (FPA)
- Synchronous backplane interconnect (SBI) bus
- MS780-E memory controller and memory array modules
- DW780 UNIBUS adapter (UBA) and peripheral devices
- RH780 MASSBUS adapter (MBA) and peripheral devices

1.2.1 Console Subsystem

An LSI-11 microcomputer provides the operator interface between the console terminal and the CPU and performs console memory and I/O operations on its internal, quad-size data/address bus (Q bus). On a power-up sequence, it loads the console program and the CPU microprogram (microcode) from the console flexible diskette (floppy disk). With a complete microengine in place, it is then able to maintain communications and operations between the console and the CPU.

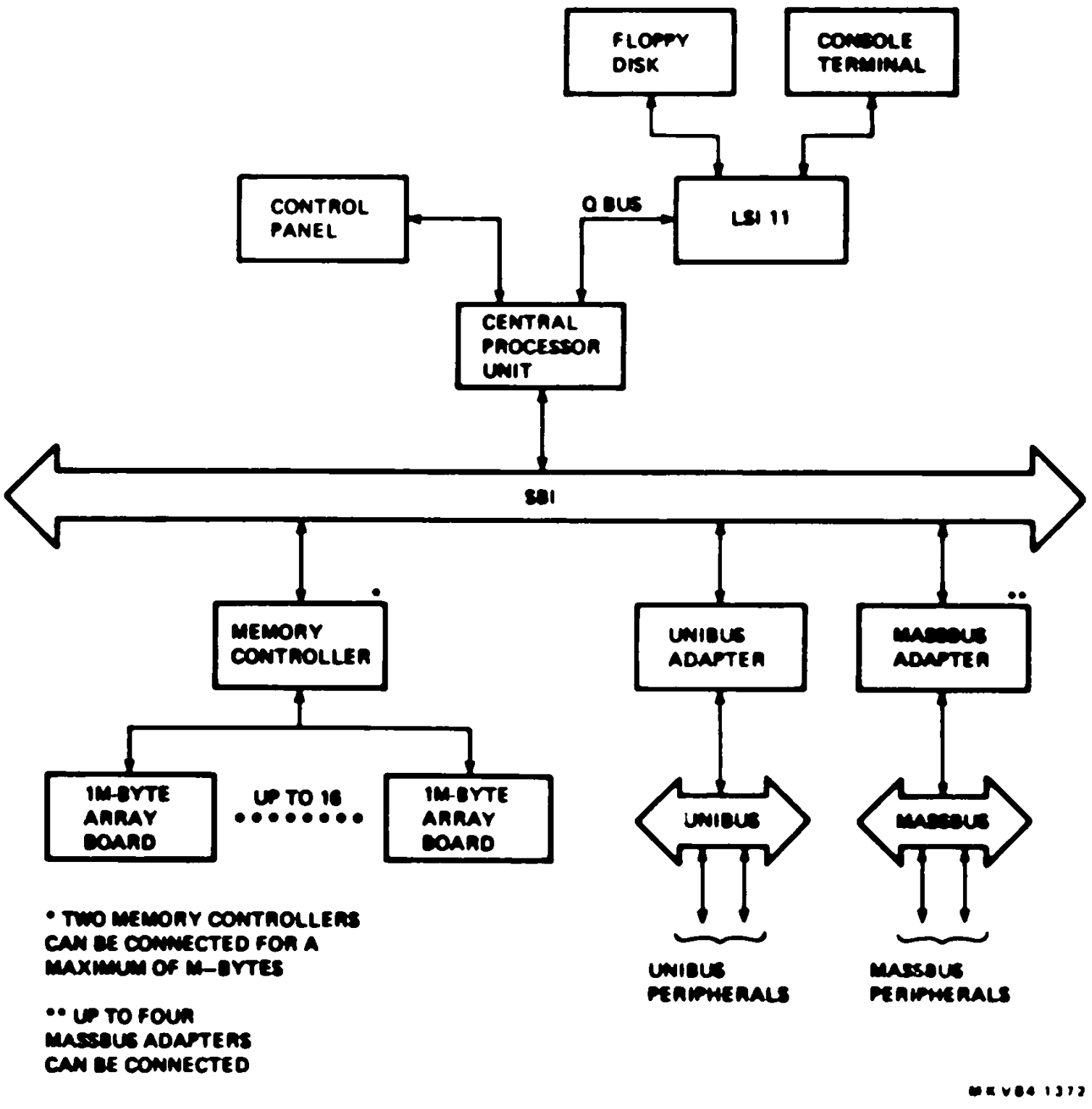


Figure 1-1 VAX-11/785 System Block Diagram

1.2.2 Central Processor Unit (CPU)

The CPU operates under the control of a high-speed, synchronous microprocessor that executes a comprehensive set of variable length machine language (MACRO) instructions in the native mode It also features a compatibility mode that executes nonprivileged PDP-11 instructions, allowing most existing PDP-11 user programs to run without modification

1.2.3 Floating-Point Accelerator (FPA)

The FPA option is an extension of the CPU logic that improves system performance by executing the floating-point instructions, reducing their execution time

1.2.4 Synchronous Backplane Interconnect (SBI)

The SBI passes parity checked, parallel transfers between the CPU, main memory, and all SBI devices Each device or subsystem on the SBI operates on event times that are synchronized with the CPU (events occur at defined points in time during CPU and SBI cycles).

The SBI provides a physical address space of over one billion bytes which represents all possible memory and I/O addressing available to the processor Physical address space is divided in half with the upper half assigned to I/O devices and registers The lower half is reserved for main memory and future memory expansion Refer to the *VAX-11/785 Translation Buffer, Cache and SBI Control Technical Description* (EK-MM785-TD) for a description of the SBI

1.2.5 Main Memory Subsystem

Main memory consists of a controller and from one to sixteen MOS RAM memory array modules The controller includes error checking and correction (ECC) logic that detects and corrects single bit errors, and detects and reports double bit errors

Each MS780-E memory controller can access a maximum of 16 million bytes (16M bytes) Two memory controllers may be connected to the SBI to provide up to 32 million bytes (32M bytes) of physical main memory space

1.2.6 UNIBUS Adapter (UBA)

The DW780 provides an interface for UNIBUS controllers and I/O devices including terminals, line p ...ters, and card readers, and allows the CPU to access UNIBUS device registers It performs priority arbitration for devices on the UNIBUS and allows them to access either sequential or random main memory locations within a page

A UBA memory map stores the base address for each page of main memory that is valid for a UNIBUS device When a DMA cycle is initiated by a device, the UBA uses the map contents to translate the 18-bit virtual address asserted by the device to a 30-bit physical byte address in main memory The UBA map addressing structure allows UNIBUS devices to transfer physically contiguous blocks on the magnetic medium to or from discontiguous blocks in main memory

Buffered data paths are available for up to 15 nonprocessor request (NPR) devices Each data path has a 64-bit quadword buffer (with parity) that holds up to four 16-bit data words Only one SBI transfer is then required for every four UNIBUS transfers to or from a data path

1.2.7 MASSBUS Adapter (MBA)

The MASSBUS adapter is a general purpose controller and interface for high-performance magnetic disk and tape drives. Up to four MBAs may be installed on the SBI. Up to eight devices may be installed on each MBA.

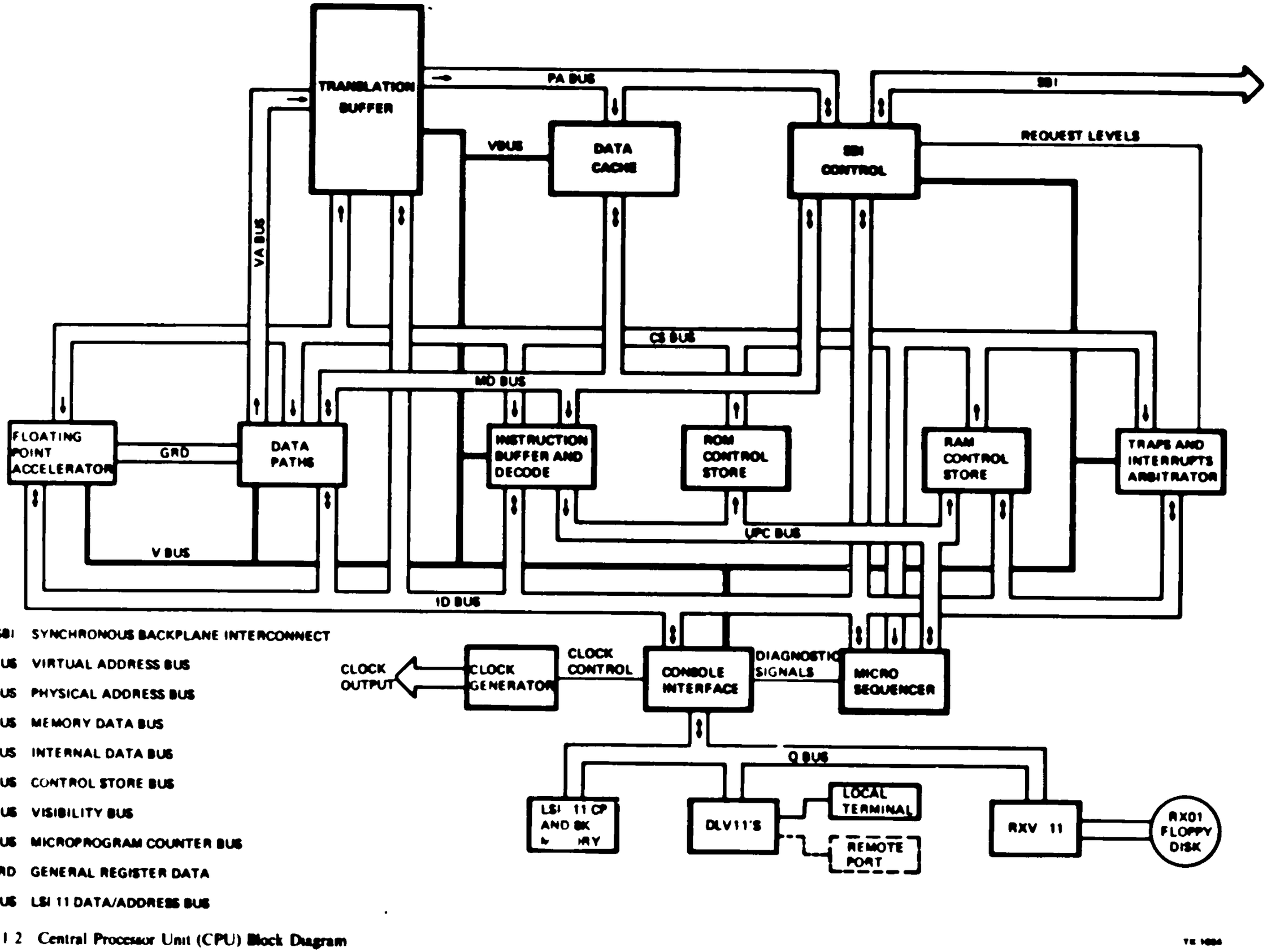
An MBA memory map stores the base address for each page of main memory that is valid for the active MASSBUS device. When a DMA cycle is initiated by the device, the MBA uses the map contents to translate the 17-bit virtual address held by an MBA register to a 30-bit physical byte address in main memory. The MBA map addressing structure allows MASSBUS devices to transfer physically contiguous blocks on the magnetic medium to or from discontiguous blocks in main memory.

A 32-byte silo data buffer (SILO) makes use of the wider SBI bandwidth, allowing efficient transfers of data between main memory and a MASSBUS device. The MBA SILO logic assembles four words of MASSBUS data into a 64-bit quadword (plus parity). Only one SBI transfer then takes place for every four MASSBUS transfers to or from the SILO.

1.3 CPU INTRODUCTION

Figure 1-2 identifies the following CPU logic elements in the KA785 backplane and shows the interconnecting busses.

- Floating-point accelerator (FPA)
- CPU data paths
- Instruction decoder (IRD)
- CPU microprocessor (microsequencer and control store ROM and RAM)
- Exceptions, interrupts, and SBI arbitration
- System clock (CLK) generator
- Console interface board (CIB) to the LSI-11
- Translation buffer (TB), cache memory, and SBI control



SBI	SYNCHRONOUS BACKPLANE INTERCONNECT
VA BUS	VIRTUAL ADDRESS BUS
PA BUS	PHYSICAL ADDRESS BUS
MD BUS	MEMORY DATA BUS
ID BUS	INTERNAL DATA BUS
CS BUS	CONTROL STORE BUS
V BUS	VISIBILITY BUS
UPC BUS	MICROPROGRAM COUNTER BUS
GRD	GENERAL REGISTER DATA
Q BUS	LSI 11 DATA/ADDRESS BUS

Figure 1 2 Central Processor Unit (CPU) Block Diagram

1.3.1 Floating-Point Accelerator (FPA)

When in place, the FPA option performs the add, subtract, multiply, and divide instructions that operate on single- and double-precision floating-point operands, including the EMOD and POLY instructions

The FPA is not required for the processor to execute floating-point instructions and may be added to or removed from the system without changing the existing software For a description of the FPA, refer to the *FP785 Floating-Point Accelerator Technical Description* (EX-FP785-TD)

1.3.2 CPU Data Paths

The data path section performs arithmetic and logical operations on data It contains the arithmetic logic unit (ALU) and data and shift registers The D and Q registers hold and pass data to and from the rest of the CPU over the ID bus.

1.3.3 Instruction Buffer and Decoder

Under control of the CPU microprogram, the instruction fetch logic fetches an instruction from main memory and stores it in the instruction buffer The microprogram then executes the instruction using the operation code and operand specifier interpretation supplied by the decoder

1.3.4 CPU Microprocessor

The microsequencer addresses, fetches, and executes microinstructions from the control store Two joint control store modules make up the writable control store (WCS) and PROM control store (PCS) sections of the CPU microsequencer

The WCS stores the CPU microprogram (microcode) and consists of 7 5K × 99-bit microwords of RAM This includes three parity bits that are generated and stored by the microsequencer logic when the WCS is written by the console on a power-up sequence

The PCS consists of 0 5K × 99-bit microwords of ROM, including three parity bits. The PCS contains basic microcode routines that initialize and halt the CPU and allow the operator to examine or deposit machine registers and memory It also contains the entry points for the microtrap and console service routines.

1.3.5 Exceptions, Interrupts, and SBI Arbitration

Exceptions and interrupts are the methods by which the CPU services system events and conditions Both types cause the processor to leave normal program flow for service

Exceptions are generated in and are synchronous to the CPU Three types of exception conditions are detected by the microcode or CPU logic traps, faults, and aborts

Interrupts may be generated internal or external to the CPU or by the software Interrupts are asynchronous to the CPU and are synchronized by the microcode.

For a device on the SBI to perform a direct memory access (DMA) transfer or receive service by an interrupt request, it must first gain access to the SBI through priority arbitration A higher-priority device is allowed access over a lower-priority device; access is granted to the lower-priority device when the first has completed its transaction

1.3.6 System Clock (CLK) Generator

The CLK module provides the time base for the CPU microprocessor and generates the CPU and SBI time states. The CPU operates on a 50% faster cycle time than the SBI using Fairchild Advanced Schottky TTL (FAST) logic, emitter-collector logic (ECL) and 100K ECL (LECL) logic. The faster CPU cycle reduces the time required by the microprocessor to execute machine language instructions and to access the TB and cache.

The CPU also contains two other clocks that are used by the VAX/VMS operating and diagnostic systems:

- *Interval timer* times system events and functions
- *Time-of-day clock* provides time of day, date, and year information. A standard battery back-up maintains the information during a power down or power failure period

1.3.7 Console Interface Board (CIB)

The LSI-11 interfaces the CPU through the console interface board (CIB). The CIB passes register information and transfers in either direction are initiated by interrupt. For a description of CIB registers and functions, refer to the *VAX-11/785 Console Interface Technical Description* (EK-KC785-TD)

1.3.8 TB, Cache, and SBI Control

The following logic is described in the *VAX-11/785 Translation Buffer, Cache and SBI Control Technical Description* (EK-MM785-TD).

- *Translation Buffer* (TB) stores physical main memory page addresses that are valid for the current process. The memory management logic uses this information to translate virtual addresses referenced by the CPU to physical addresses in main memory. Each address stored in the TB also contains access protection information for the page. The TB provides 512 (.5K) RAM locations.
- *Cache Memory* (cache) stores program and data information from the most recently accessed memory locations, reducing the average time required by the CPU to retrieve main memory information. The cache provides 32K bytes of RAM locations.
- *SBI Control* interfaces the CPU to the SBI data bus and provides the synchronization and timing for all adapters and subsystems in the backplane, including the CPU. The data/address lines transfer command/address and data information between main memory and the CPU and other SBI devices.

1.4 READING THE CPU MICROCODE

The CPU microprogram (CPU microcode) performs the operations and functions that execute the VAX-11 native mode and PDP-11 compatibility mode instructions. The CPU microprocessor sequences and interprets the microprogram using a 96-bit control word called a microword or microinstruction. Figure 1-3 shows the microword bit field format and lists the defined values for each field.

The microassembler accepts, as inputs, definitions that it uses to format the machine language (macro) instructions. This section briefly describes the following definitions that are used to express values and functions in the microcode listing:

- Field definitions
- Value definitions
- Microinstruction definitions
- Comments
- Macros
- Microaddress labels
- Control store memory allocation

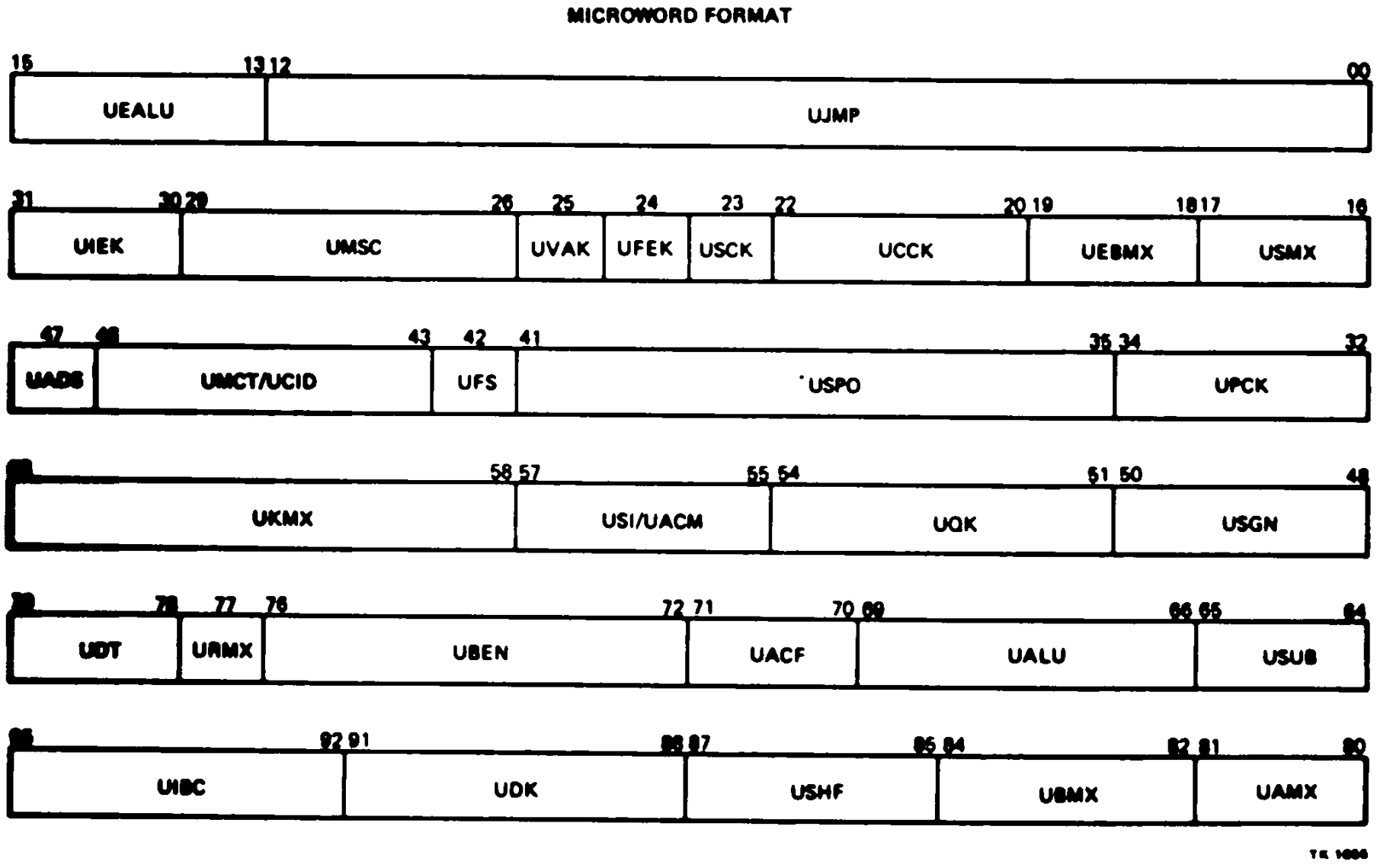


Figure 1-3 Microword Format and Field Definitions
(Sheet 1 of 5)

MICROWORD < 31:00 >

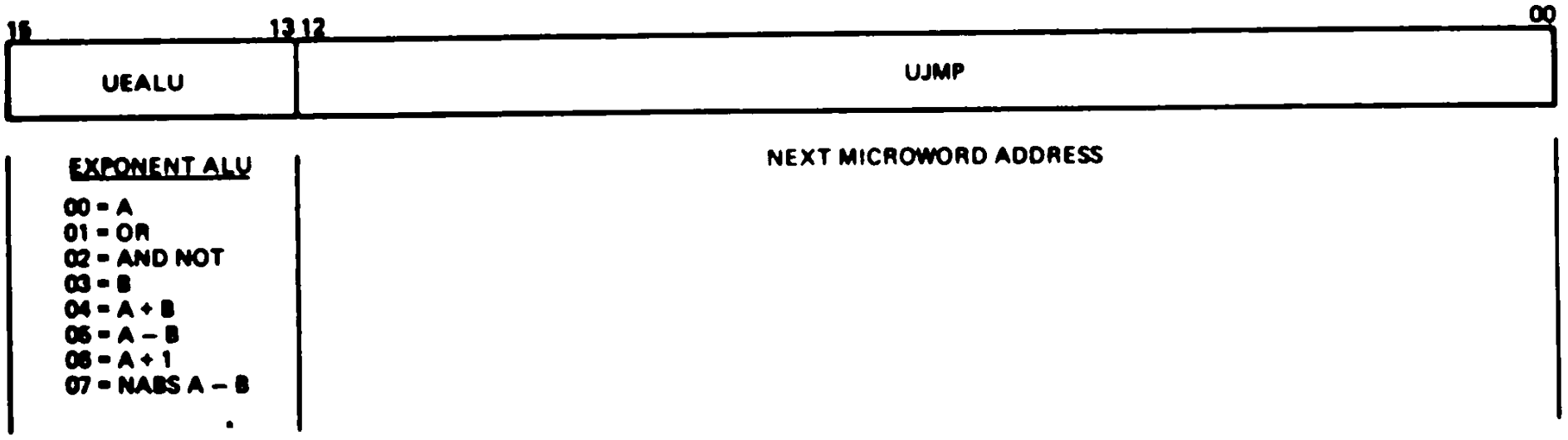


Figure 1-3 Microword Format and Field Definitions
(Sheet 2 of 5)

MICROWORD < 47:32 >

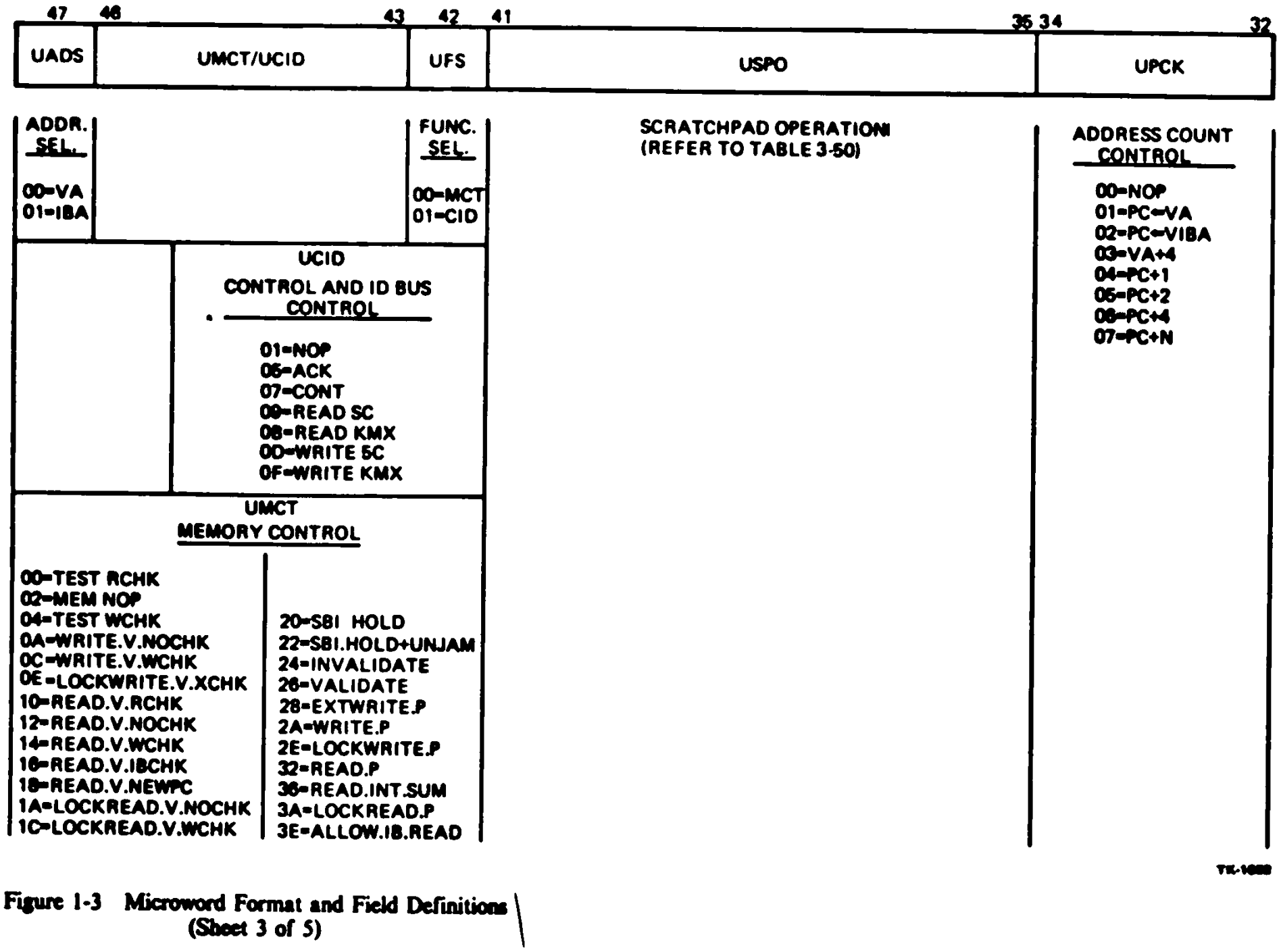


Figure 1-3 Microword Format and Field Definitions
(Sheet 3 of 5)

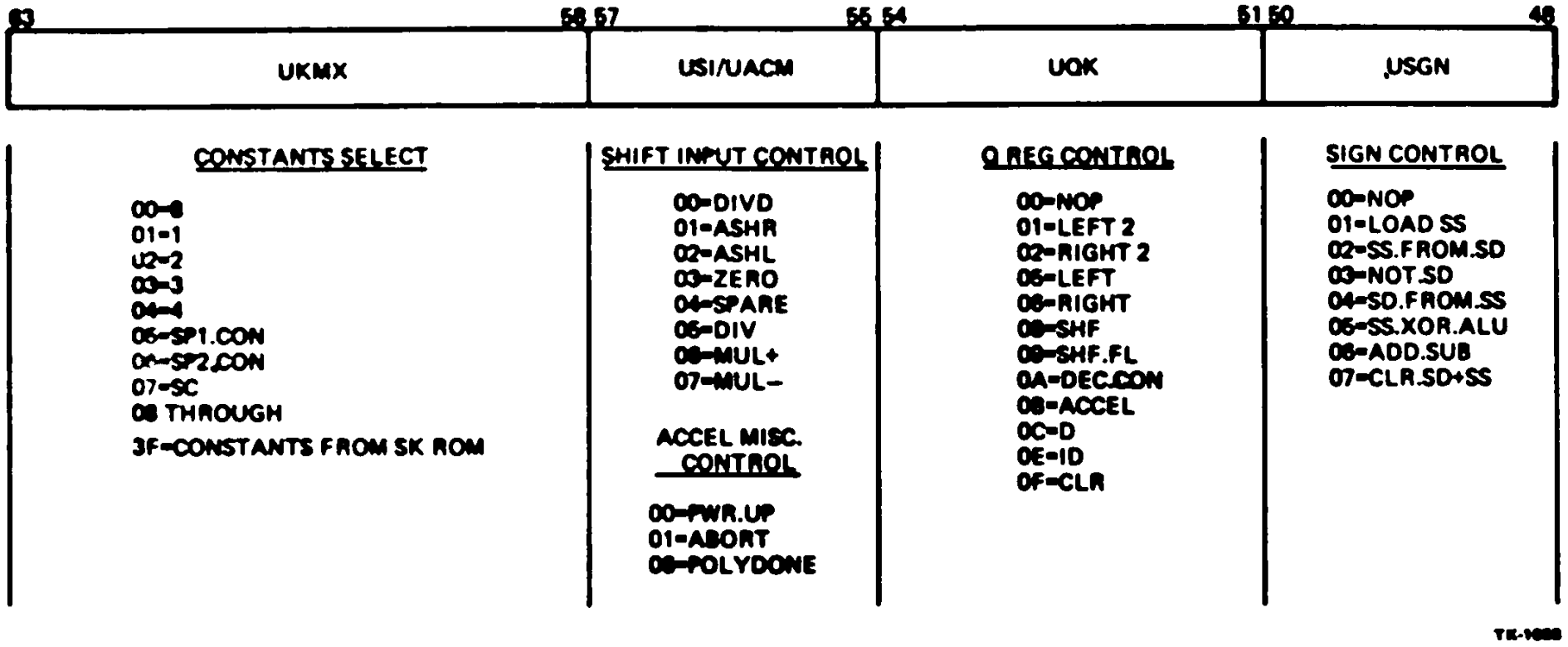
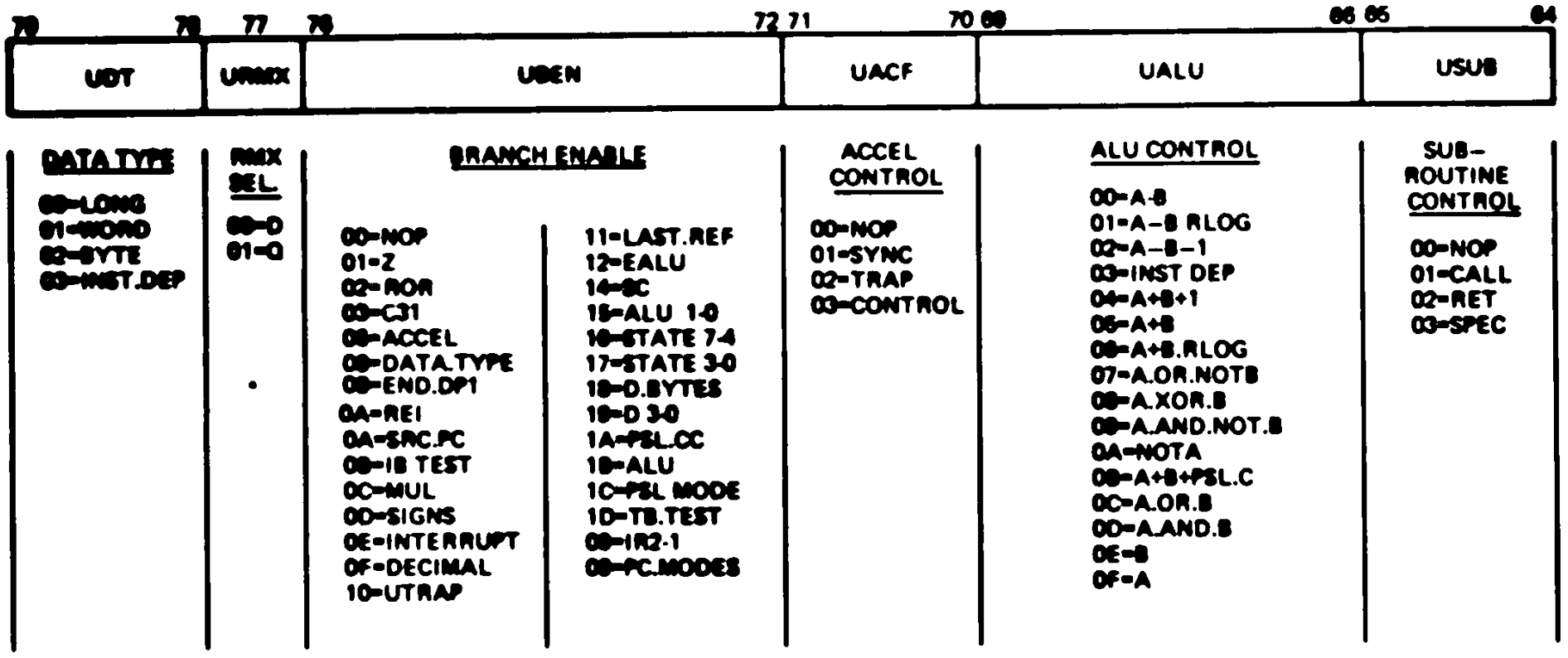


Figure 1-3 Microword Format and Field Definitions
(Sheet 4 of 5)

MICROWORD < 95:64 >



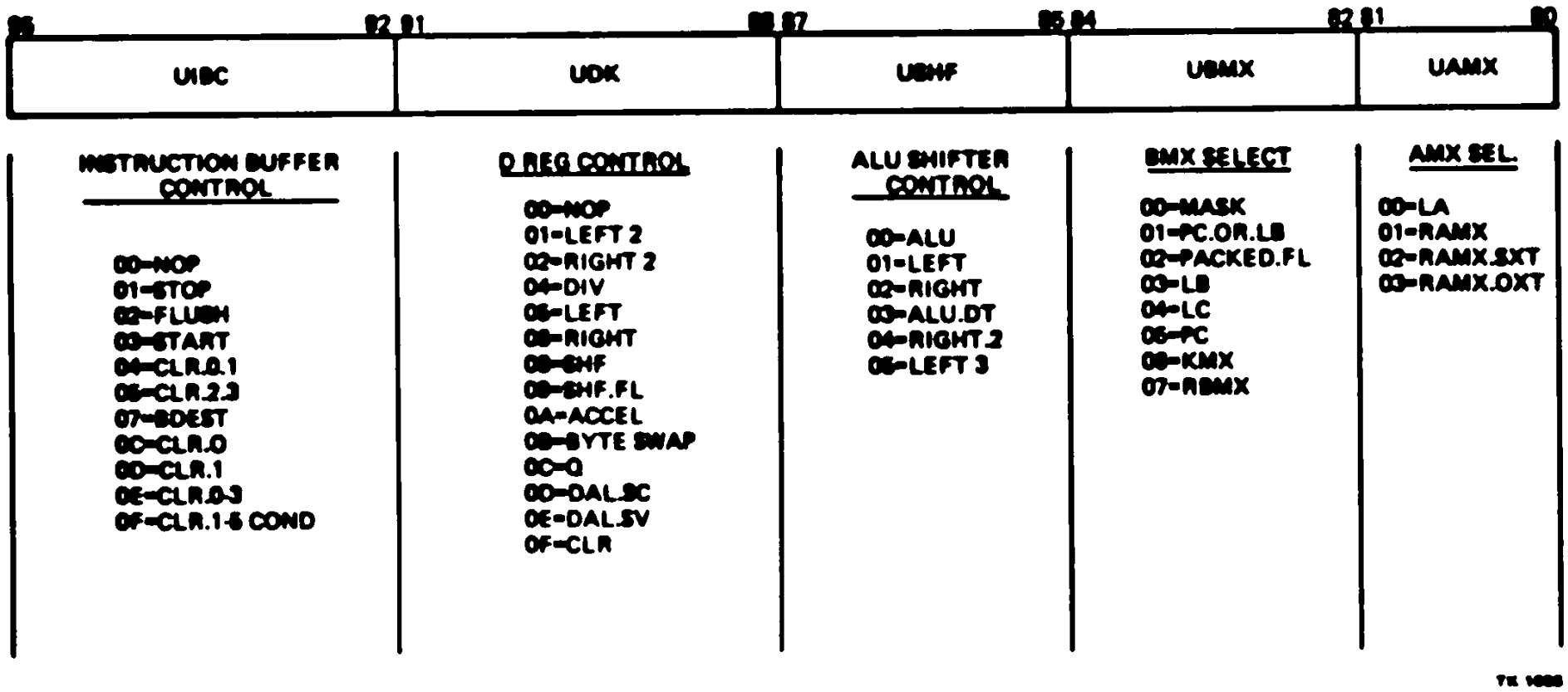


Figure 1-3 Microword Format and Field Definitions
(Sheet 5 of 5)

1.4.1 Field Definitions

Each microword bit field is defined in the following format

field_name/=(high_bit low_bit)

field_name	-	functional name of the field
high_bit	-	starting bit number of the field
low_bit	-	ending bit number of the field

For example, the arithmetic logic unit (ALU) control field (bits 66 through 69 of the microword) is defined as follows

```
ALU/=(69:66)
```

To specify a default value for a microword field, the " DEFAULT" pseudo-operator immediately follows the field definition

```
ALU/=(69 66), DEFAULT=0
```

In this example, the value of the ALU field is zero unless otherwise specified

1.4.2 Value Definitions

Symbols are then used to specify the value that a microword field may contain and are defined on the lines following the field definition For example, several ALU field values may be defined as follows

```
ALU/=(69:66), DEFAULT=0
        A-B=00
        A-B RLOG=01
        A-B-1=02
        INST DEP=03
        A+B+1=04
```

When used in reference to the ALU field, the symbol "A-B-1", has a value of two and the symbol "A+B+1" has a value of four

1.4.3 Microinstruction Definition

In each word of microcode, each field is assigned a specific value To assign a value to a microword field, its symbolic name is used, followed by a slash (/), followed by the value that the field is to contain

If multiple fields are to be defined in the same microword, the field definitions are separated by commas as follows

```
AMX/RAMX, BMX/LB, ALU/A-B
```

This microword assigns the value RAMX (1) to the AMX field (bits (81 80)), assigns the value LB (3) to the BMX field (bits (84 82)), and assigns the value A-B (0) to the ALU field (bits (69 66))

Field definitions may be continued on more than one line The following microword is identical to the one above

```
AMX/RAMX, BMX/LB,
ALU/A-B
```

If a line of microcode ends in a comma, that microword is continued through the next line If there is no comma at the end of a line, that is the end of that microcode word

1.4.4 Comments

When a semicolon appears on a line, all text from the semicolon through the end of the line is considered a comment and is ignored by the microcode assembler

A semicolon followed by a line of dashes is often used in the microcode listings to make it easy to see the microword boundaries

1.4.5 Macros

In the microcode, even simple tasks may require setting many different fields For example, to add the Q and D registers together and place the sum in the D register, the following steps are necessary

```
RAMX/Q, AMX/RAMX, RBMX/D, BMX/RBMX,
ALU/A+B, SHF/ALU, DK/SHF
```

Because so many fields are defined in this example, it is not easy to see what function the microword is supposed to perform Therefore, a symbol is assigned to a group of field definitions This symbol is called a macro

The macro "D__Q+D" is defined for the seven field definitions in the above example Then, whenever the macro "D__Q+D" is used, each of the seven fields is set to the values specified in the macro definition In macro names, the underscore character (__) is used as an assignment operator

Macros are defined in the MACRO MIC section of the microcode listing Many macros may be used in one microword by separating the macro names with commas Macros and field definitions may be used together in the same microword

Macros may have parameters that are specified in the macro name by a value enclosed in brackets ([ ]) In the macro definition, an "at" character (@) is used, followed by a decimal integer, to show where the parameter belongs For example

```
ALU__R[ ]-K[ ]        SPO R/LOAD LAB, SPO RAB/@1, AMX/LA,
                      KMX/@2, BMX/KMX, ALU/A-B
```

If this macro were used as follows

```
ALU__R[R4]-K[ZERO]
```

the SPO RAB field would then have the value "R4" and the KMX field would have the value "ZERO"

1.4.6 Microaddress Labels

Symbolic labels may be assigned to the address of a microword. An address label is a symbol followed by a colon that precedes a microword as in this example:

```
DEC MUL     D__0-D,
            J/DEC L2P1
```

The symbol "DEC MUL" is defined as the address of that microword.

Each label in the microcode listing has a two- to four-character prefix that corresponds to the section of microcode in which it is defined. In the previous example, "DEC MUL" is defined in the decimal string microcode section. The label "QUE FAIL 2" is defined in the queue instruction microcode section. The label "GH ALNPOSG" is defined in the microcode dealing with G__ and H__floating instructions.

A hex number may be used as a label to assign a microword to a particular location in control store memory. For example:

```
0E00
INI SYS INIT
        SC__K[F],
        STOP IB, MCT/MEM NOP
```

The microword shown above is at microaddress 0E00. Note that a symbolic label may also be used when a microaddress label is specified.

1.4.7 Control Store Memory Allocation

Low-order microaddress bits may be specified by using constraint blocks. A constraint block allocates a series of microwords having a particular low-order bit pattern.

A constraint block is defined as the character "=" at the beginning of a line, followed by a string of zeros, ones, and/or asterisks. The number of characters following the equal sign is the number of low-order bits in the microaddress to be specified.

For example, the constraint block definition "=10011" tells the assembler to allocate four microaddresses whose low five bits will be the following:

```
10011
10111
11011
11111
```

Any unspecified bits in the microaddress are the same for all microaddresses in the constraint block. A typical set of microaddresses for the constraint block "=10011" might be:

```
1 1001 1111 0011
1 1001 1111 0111
1 1001 1111 1011
1 1001 1111 1111
```

The only bits in the microaddresses in a constraint block that change are the bits that contain zeros in the constraint block definition.

An asterisk in a constraint block definition is a nonapplicable condition and the microassembler places the same value in all of the microwords at that position. For the constraint block "=1*0", a possible set of microaddresses might be

```
1 1001 1111 1100
1 1001 1111 1101
```

or

```
0 1110 0101 0110
0 1110 0101 0111
```

Here, the address progression in the constraint block is counting in the "0" positions of the constraint string. To skip over unneeded locations, a new constraint block may be used. For example, if the third location in the "=10011" example were not needed, then

```
=10011    (first microword)
          (second microword)
=11111    (third microword)
```

A null constraint block definition ("=" not followed by any characters) terminates an open constraint block.

1.5 SYSTEM LAYOUT AND CPU MODULE UTILIZATION

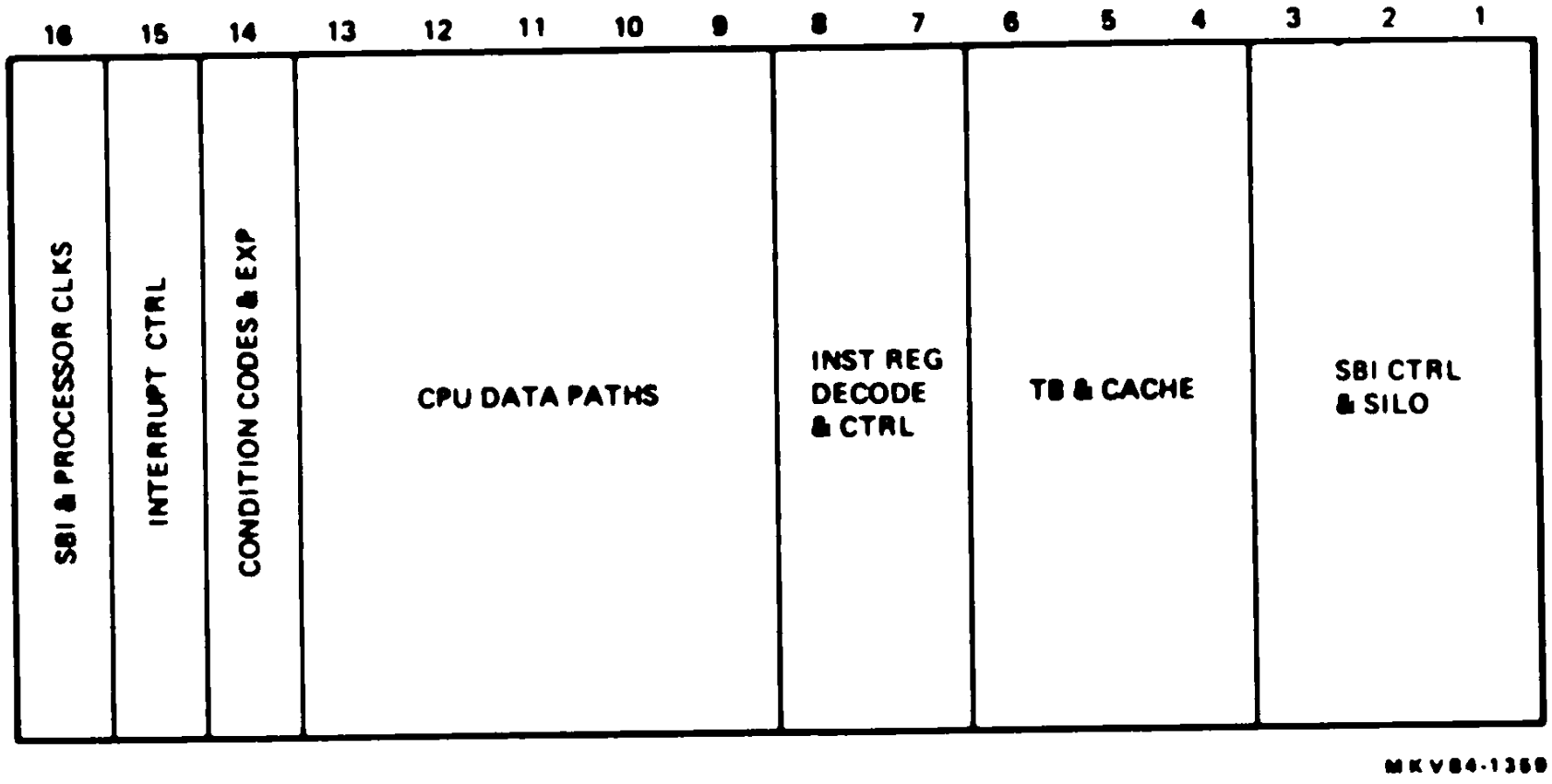
Figure 1-4 shows the logic backplane and hardware layout inside the front doors of the system cabinet. The VAX-11/785 typically contains the hardware listed in Table 1-2.

Table 1-2 Typical VAX-11/785 Hardware Layout

Option	Description
KA785	Central processor unit (CPU)
MS780	Main memory controller and MOS RAM array modules
DW780	UNIBUS adapter (UBA)
RH780	MASSBUS adapter (MBA 0, optional)
RH780	MASSBUS adapter (MBA 1, optional)
H7100A	System power supplies
H7112A	Main memory battery backup (optional)
H7111A	Time-of-day clock (TODC) battery backup
869B	System power controller (not shown)
LSI-11	Console microcomputer (front end)
RX01	Console flexible diskette (floppy disk) drive
LA120	Console terminal (not shown)

Figure 1-5 shows the logical distribution of modules in the KA785 backplane. Table 1-3 provides a list of the CPU modules, names, and locations.

Figure 1-4 VAX-11/785 System Cabinet Layout, Typical Front View


16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
SBI & PROCESSOR CLKS
INTERRUPT CTRL
CONDITION CODES & EXP
CPU DATA PATHS
INST REG DECODE & CTRL
TB & CACHE
SBI CTRL & SILO
MKV84-1359

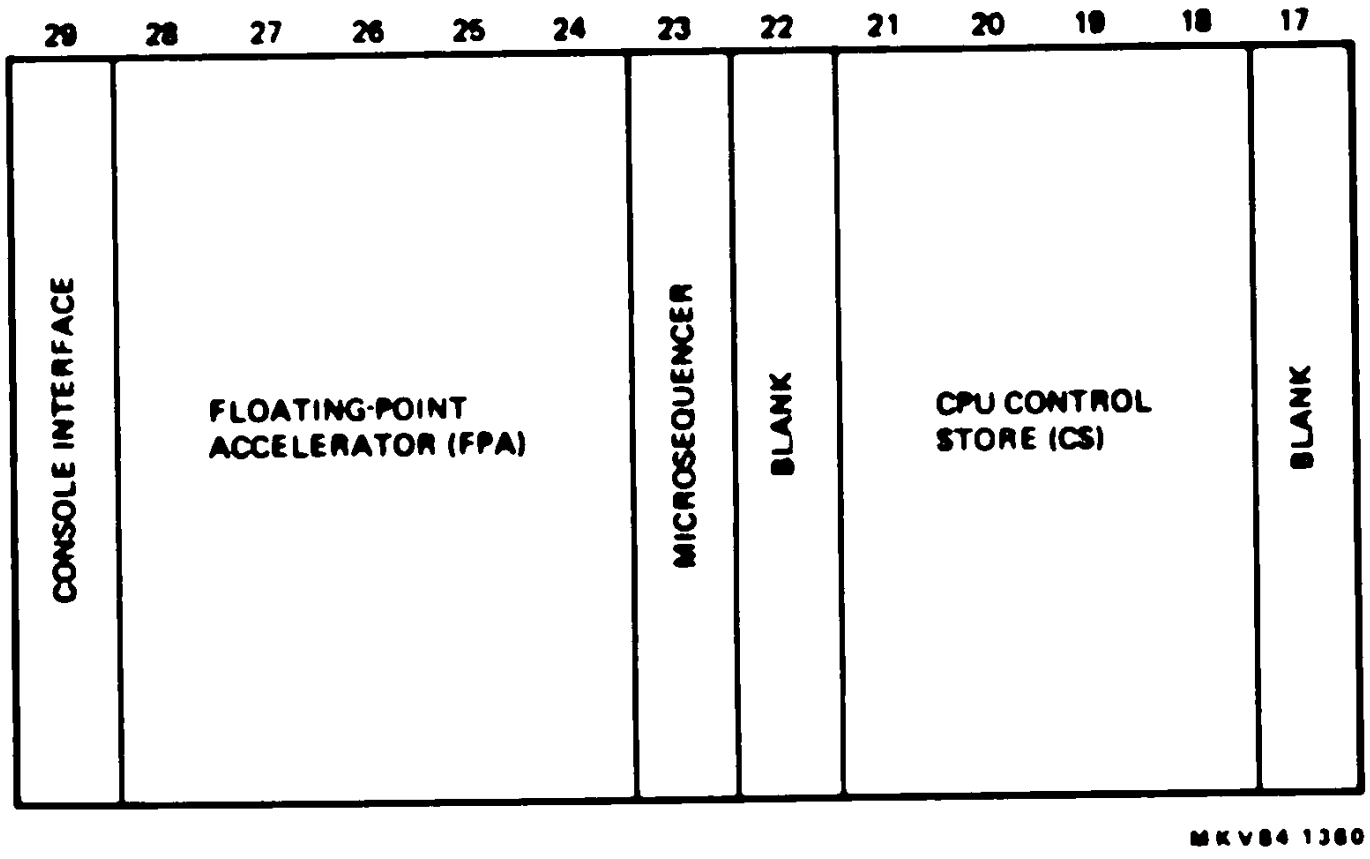


Figure 1-5 CPU Backplane Logic Distribution

Table 1-3 KA785 Module Utilization

Module Number	Mnemonic	Module Name or Function	Slot Location
M7477	CIB	Console interface board	29
M7544	FCT	FPA control	28*
M7543	FAD	Fractional adder	27*
M7542	FML	Fraction multiply - low	26*
M7541	FMH	Fraction multiply - high	25*
M7540	FNM	Fractional normalizer	24*
M7476	USC	Microsequencer	23
		Blank module	22
		Open slot	21
M7475	JCS	Joint Control Store	20
		Open slot	19
M7475	JCS	Joint Control Store	18
		Open slot	17
M7474	CLK	Processor clocks	16
M7473	ICL	Interrupt control	15
M7472	CEH	Condition codes and exceptions	14
M7471	DAP	Data path control	13
M7470	DCP	Data path - (07:00)	12
M7469	DDP	Data path - (15:08)	11
M7468	DEP	Data path - (31 16)	10
M7467	DBP	Data path - multiplexers/shifters	9
M7466	IRC	Instruction decoder	8
M7465	IDP	Instruction data path	7
M7464	TBM	Translation buffer matrix	6
M7463	CDM	Cache data matrix	5
M7462	CAM	Cache address matrix	4
M7461	SBH	SBI interface - high bits	3
M7460	SBL	SBI interface - low bit	2
M7459	TRS	SBI terminator and silo	1

* If the FPA is not present, these slots contain blank modules to maintain proper airflow.

CHAPTER 2

CHAPTER 2
FUNCTIONAL DESCRIPTION

2.1 SYSTEM ARCHITECTURE

The VAX-11/785 is an extension of the PDP-11 family of computers and is compatible with the PDP-11 line of hardware peripherals and options. Virtual address extension (VAX) techniques use magnetic disk as the main storage device in conjunction with system main memory in order to make large amounts of apparent memory space available to the programmer and user.

2.1.1 Virtual and Physical Addressing

The VAX/VMS operating system executes procedures that are able to map (allocate) over four billion bytes of virtual address space into one billion bytes of physical address space.

The CPU memory management logic performs the actual translation of virtual addresses to physical SBI addresses and provides the capabilities for sharing, paging, swapping, and overlay protection. For a description of the address translation process, refer to the *TB, Cache, and SBI Control Technical Description* (EK-MM78X-TD).

2.1.1.1 Virtual Address Space – Figure 2-1 shows virtual address space allocation which is divided into two parts: system virtual space and process virtual space.

The lower half of system virtual space is designated as system space (S0). System space is shared by all processes and is not changed on context switches. The upper half is reserved for future use.

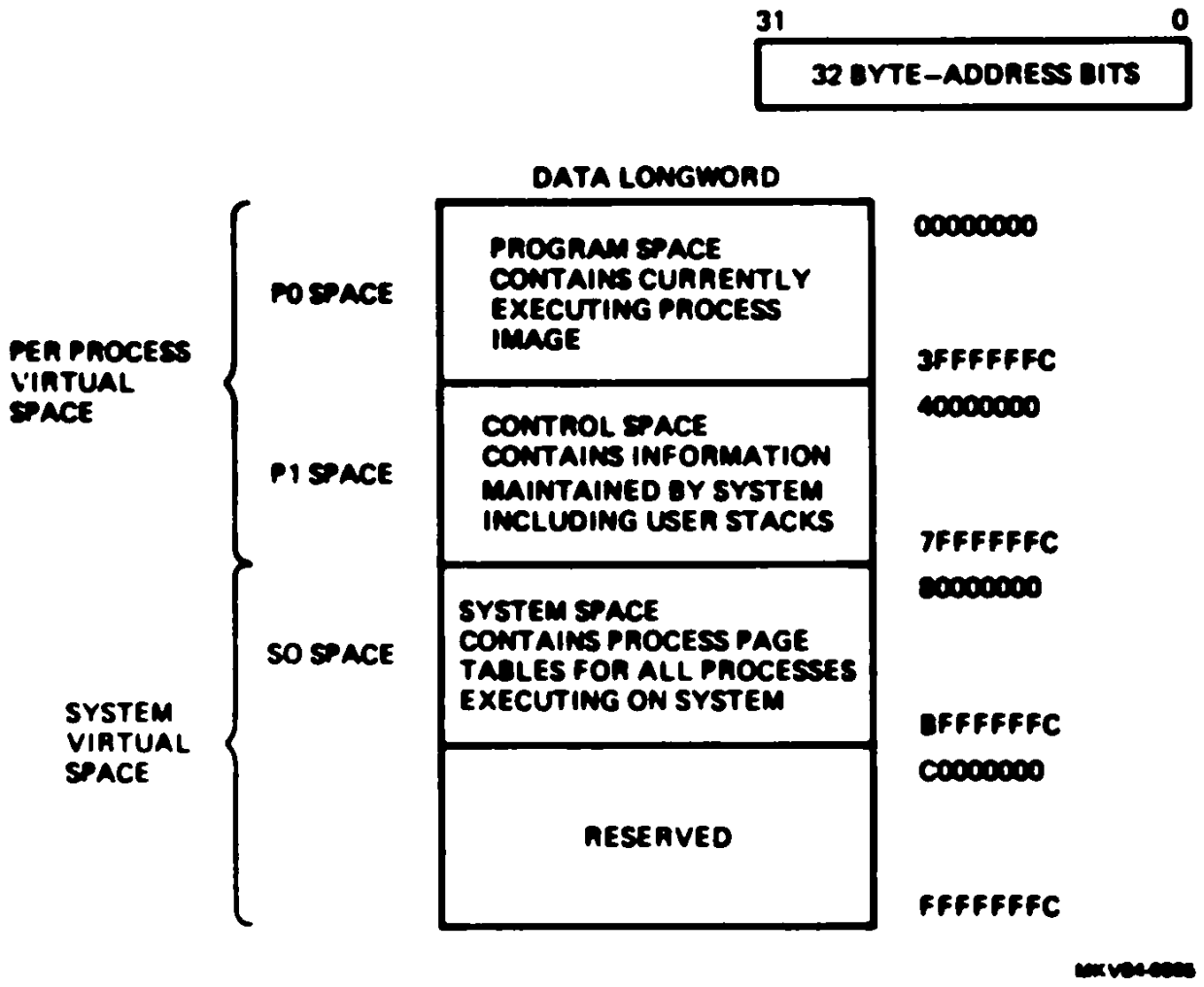


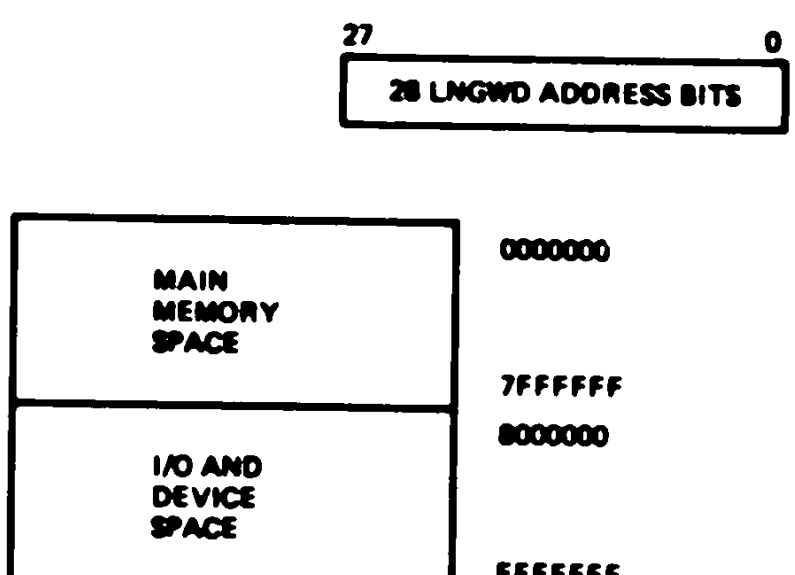
Figure 2-1 Virtual Address Space

Per-process virtual space is designated for use by each process. A context switch changes the mapping of all addresses in process space to accommodate the current process. Per-process space is also divided into two halves. Program space (P0) contains the process image currently executing on the system. Control space (P1) contains the user stacks and I/O buffers for the currently executing process.

2.1.1.2 Physical Address Space – Figure 2-2 shows physical address space allocation, which is divided into two parts. The upper half is assigned to device input/output (I/O) and control registers and addresses. The lower half is reserved for main memory and future expansion.

On each program reference to memory, the CPU memory management logic translates a 32-bit virtual byte address to a 28-bit physical longword address that corresponds to an actual location in main memory.

The SBI provides physical address space for up to one billion bytes. This represents all possible memory and I/O address space that is available to the processor.



MKV84-0888

Figure 2-2 Physical Address Space

2.1.1.3 Data Types – The native instruction set operates on byte, word, longword, quadword, and octaword data types. However, the basic VAX addressing unit is the byte. The 32-bit virtual address identifies a byte location and the data type specified by the instruction identifies the number of bytes to be processed in the operation.

The native instruction set performs operations on the following data types and sizes:

- Integer Data
  - Byte (8 bits)
  - Word (16 bits)
  - Longword (32 bits)
  - Quadword (64 bits)
  - Octaword (128 bits)

- Floating-Point/Double-Floating Data
  - Longword (32 bits)
  - Quadword (64 bits)
  - Octaword (128 bits)
- Data Strings
  - Character
  - Leading separate numeric
  - Trailing numeric
  - Packed decimal
  - Variable-length bit field
  - Queue data

Tables 2-1, 2-2, and 2-3 show the values and the ranges or precision represented by each of the data types. In the following descriptions, each data type is specified by its base address A (the address of the byte that contains bit ⟨00⟩)

Table 2-1 Summary of Integer Data Ranges

Type	Size	Range (decimal)	
		Signed	Unsigned
Byte	8 bits	−128 to +127	0 to 255
Word	16 bits	−32768 to +32767	0 to 65535
Longword	32 bits	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
Quadword	64 bits	-2^{63} to $+2^{63}$ 1	0 to $2^{64}-1$
Octaword	128 bits	-2^{127} to $+2^{127}-1$	0 to 2^{128} 1

Table 2-2 Summary of Floating-Point Data Precision

Type	Size	Precision (Decimal Digits)
F_floating	32 bits	Approximately seven (single-precision, addressed same as longword
D_floating	64 bits	Approximately 16 (double-precision, addressed same as quadword)
G_floating	64 bits	Approximately 15 (extended-precision, same as D_floating except for exponent size)
H_floating	128 bits	Approximately 33 (extended-precision, octaword addressing with large exponent and fraction)

Table 2-3 Summary of Data String Precision

Type	Size	Precision
Numeric String	0 to 31 bytes (digits)	$-10^{31}-1$ to $+10^{31}-1$
Packed Decimal String	0 to 16 bytes (31 digits)	Two numeric digits per byte, sign in lower half of last byte
Character String	0 to 65,535 bytes (64K bytes)	One character per byte
Variable-Length Bit Field	0 to 32 bits	Interpretation dependent
Queue Data	2 or more longwords per entry	0 through 2 billion entries

2.1.2 Integer Data

Integer values are stored as signed or unsigned binary data and are stored in byte, word, longword, quadword, or, in some cases, in octaword sizes. The native instruction set processes signed integer data in two's complement arithmetic

Figure 2-3 shows the four integer data types that represent quantities in binary format Integer values are expressed in two's complement binary with bit ⟨00⟩ as the least significant bit Each quantity may be treated as signed or unsigned data The sign bit for each of the four signed integer data types is byte–bit ⟨07⟩, word bit ⟨15⟩; longword-bit ⟨31⟩, and quadword–bit ⟨63⟩

2.1.3 Floating-Point Data

A floating-point value is stored as a normalized fraction with a signed power-of-two exponent

The floating-point data types represent approximate quantities for which scaling is not specified in the program Each type is stored in scientific notation as a power of two times a (normalized) fraction in the range of 0 5 (inclusive) to 1 0 (exclusive) The format consists of three fields the power-of-two exponent, its sign, and the fractional magnitude

VAX provides two types of PDP-11-compatible floating point data single-precision (32 bits) and double-precision (64 bits), also called floating (F_floating) and double-floating (D_floating) Two extended-range formats are also available 64 bits (G_floating) and 128 bits (H_floating) The last two formats (sometimes called "grand" and "huge" for reference) have larger exponent sizes than the first two

2.1.3.1 Single-Precision Data - Figure 2-4 shows the format for a single-precision (F_floating) datum of four contiguous bytes The form of a floating datum is a sign magnitude Bit ⟨15⟩ as the sign of the the exponent and bits ⟨14 07⟩ are an excess 128 binary exponent

Bits ⟨06·00⟩ and ⟨31 16⟩ are a normalized 24-bit fraction The most significant fraction bit (the normalized 1) is discarded to provide one extra bit of precision and is accounted for in the operation The fraction bits increase in significance from bit ⟨16⟩ through ⟨31⟩, then from bit ⟨00⟩ through ⟨06⟩

The eight-bit exponent field encodes values of 000 through 255. An exponent value of 0, with a sign bit of 0, indicates that the floating datum has a value of 0. Exponent values of 001 through 255 indicate a true binary exponent of −127 through +127. An exponent value of 0, with a sign bit of 1, is a reserved operand. A floating point instruction that attempts it causes a reserved operand fault. The value of a floating datum is in the approximate range of 0.29×10^{-38} through $1.7 \times 10^{+38}$. Its precision is approximately one part in 223 (typically 7 decimal digits).

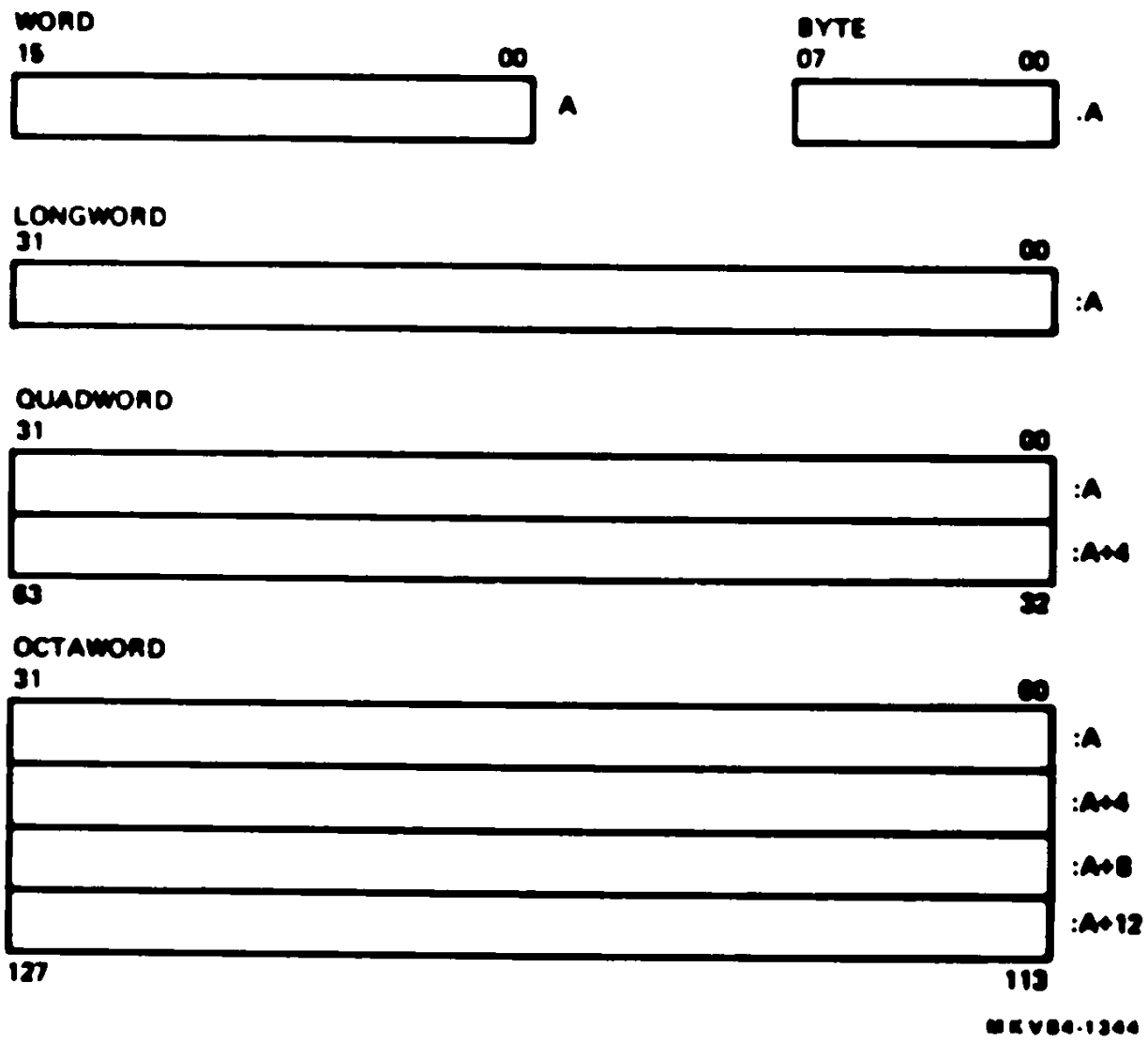


Figure 2-3 Integer Data Formats

2.1.3.2 Double-Precision Data – Figure 2-4 also shows the format for a double-precision (D_floating) datum of eight contiguous bytes. The form is the same as for F_floating except for an additional 32 lower-order fraction bits. Bit ⟨15⟩ is the sign of the exponent and bits ⟨14:07⟩ are an excess 128 binary exponent.

Bits ⟨06:00⟩, ⟨31:16⟩, ⟨47:32⟩, and ⟨63:48⟩ are a normalized 56-bit fraction. The most significant fraction bit (the normalized "1") is discarded to allow one extra bit of precision and is accounted for in the operation. The fraction bits increase in significance from ⟨48⟩ through ⟨63⟩, ⟨32⟩ through ⟨47⟩, ⟨16⟩ through ⟨31⟩, and ⟨00⟩ through ⟨06⟩. (Bit ⟨06⟩ is the highest order bit and bit ⟨48⟩ is the lowest.)

The exponent conventions and approximate range of values are the same as for floating. The precision of a double-floating datum is approximately one part in 255 (typically 16 decimal digits).

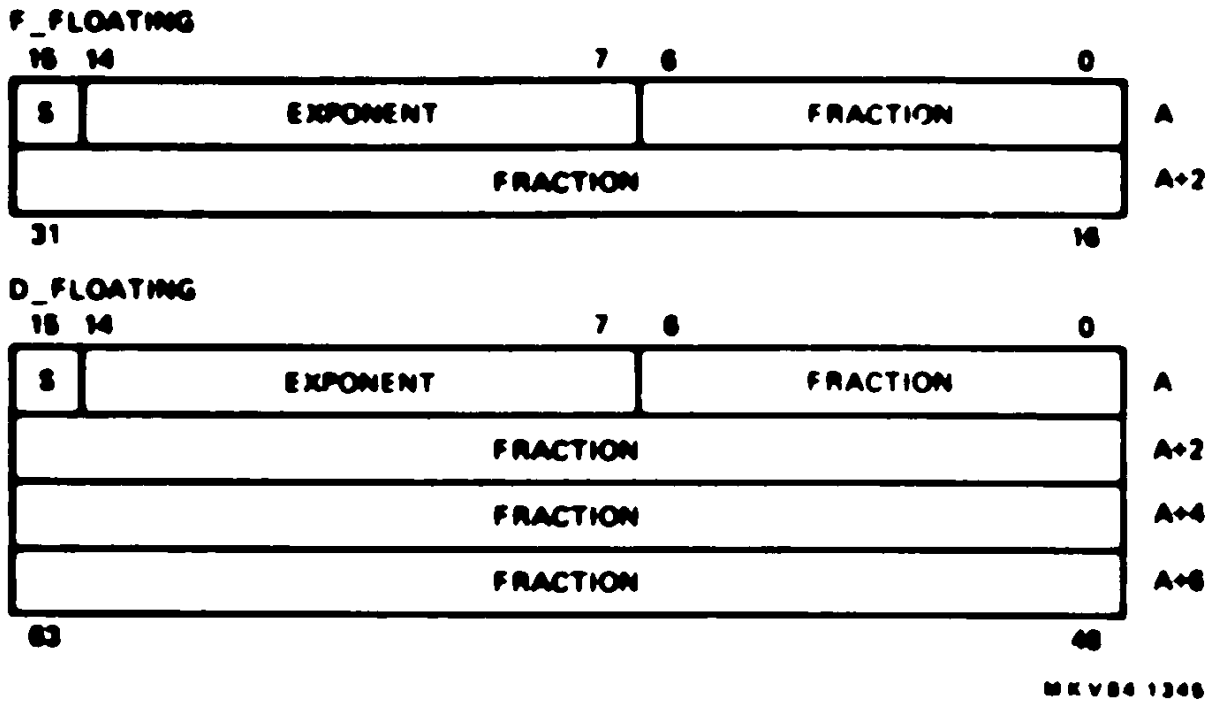


Figure 2-4 Floating/Double-Floating Data Formats

2.1.3.3 Extended-Precision Data – Figure 2-5 shows the formats for extended-precision data of eight contiguous bytes (G_floating) and 16 contiguous bytes (H_floating). G_floating data addressing is handled the same as D_floating by the logical instructions. The main difference is in the size of the exponent.

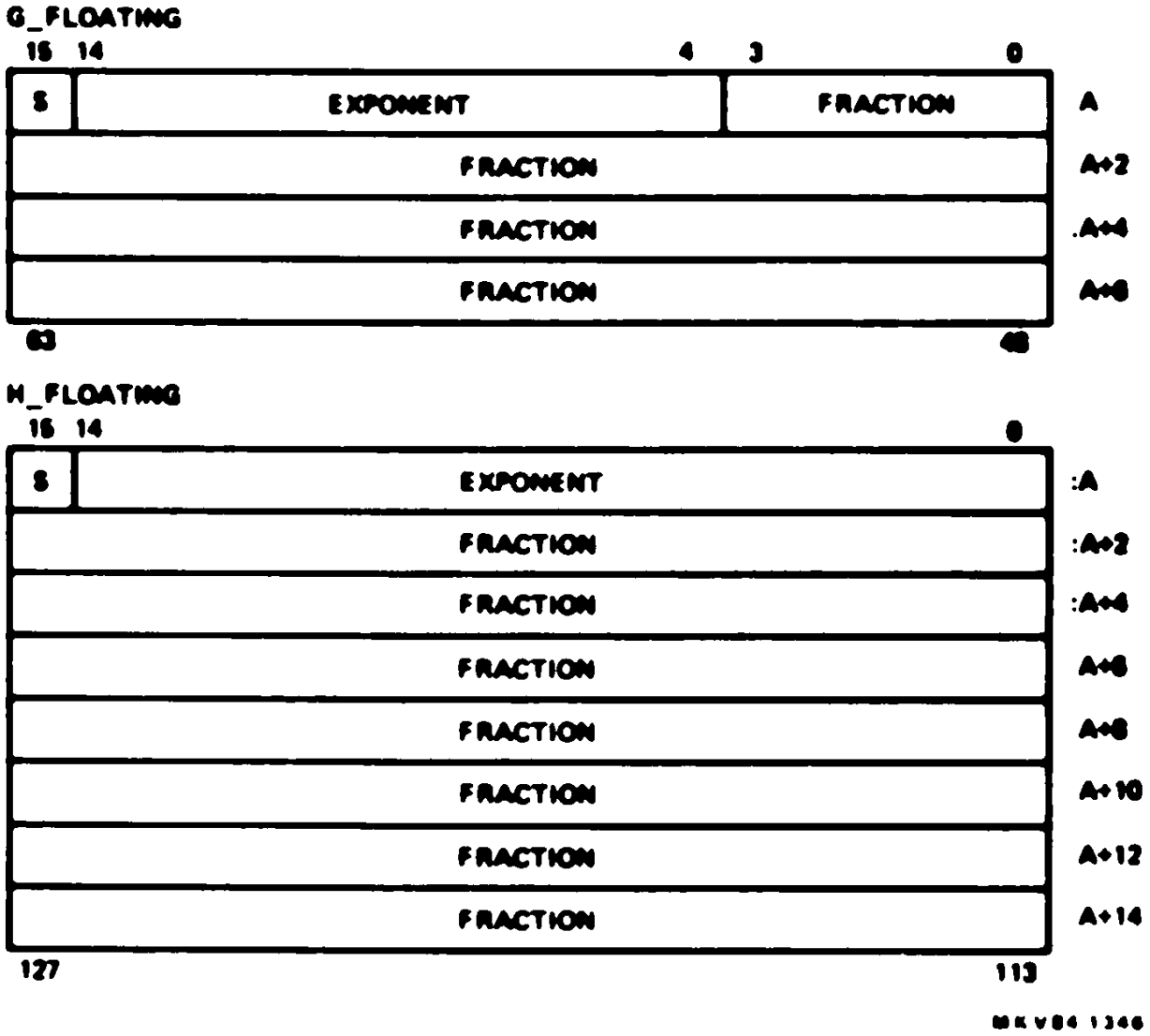


Figure 2-5 Extended-Precision Floating Data Formats

2.1.4 Data Strings

Data strings store scaled quantities in a form that resembles their external representation. They are used in routines that move or transfer information rather than perform computations on it. The data string formats occupy contiguous bytes in memory and contain the following types of information:

• Character string	ASCII alphanumeric characters or binary data
• Numeric string	ASCII numeric (decimal digit) characters with a leading or trailing sign value
• Packed decimal	Two decimal digits in two four-bit nibbles per byte with the sign in the lowest order nibble
• Variable-length bit field	A 0- to 32-bit field starting on a specified bit
• Queue data	Circular lists or tables that are forward- and backward-linked by headers

Numeric strings may represent several exact data arrangements. The main difference is whether the sign, if any, appears before the first digit (leading separate) or is encoded as part of the final digit (trailing numeric).

2.1.4.1 Character String Data – Figure 2-6 shows the character string format that may be used to represent strings of alphanumeric characters such as names, data records, or text. Typical operations include copying, concatenating, searching, or translating the string rather than performing arithmetic or logical operations on the data.

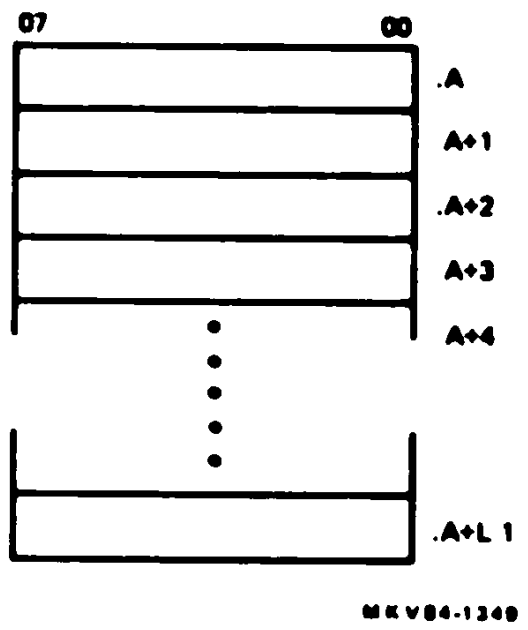


Figure 2-6 Character String Format

It is specified by base address A (the address of the first byte in the string) and by the length L of the string in bytes. The length of a single character string may be in the range of 0 through 65,535 (decimal) or 64K bytes. A string with a length of 0 is called a null string.

2.1.4.2 Leading Separate Numeric String Data – Figure 2-7 shows examples of positive and negative values in the leading separate numeric string format. Each format is specified by base address A (the address of the first byte in the string which also contains the sign character) and by the length L of the string in digits. L is not the length of the string in bytes. The actual number of bytes in a leading separate numeric string is L + 1.

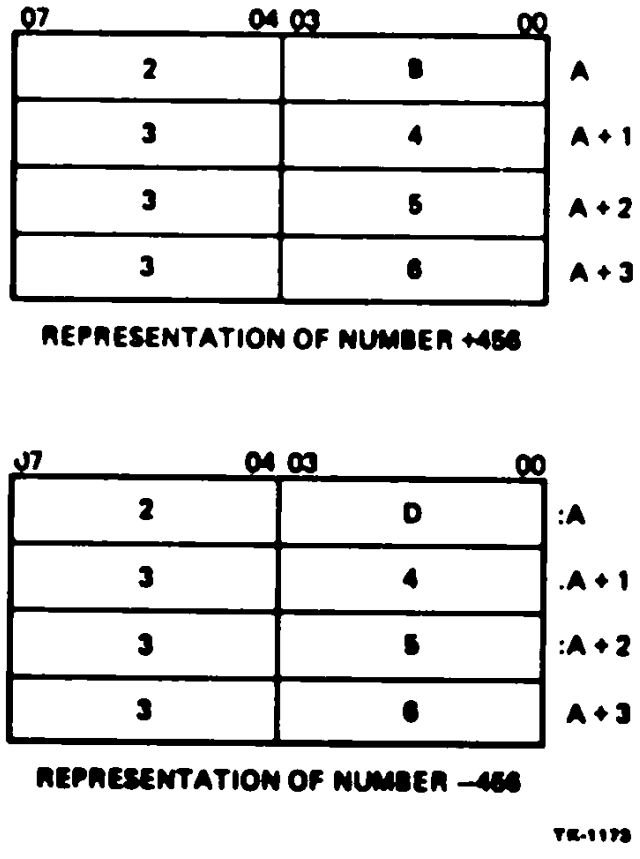


Figure 2-7 Leading Separate Numeric String Format

Table 2-4 shows the sign representations that are valid for the first byte location. They are shown in octal, decimal, and hexadecimal (hex) notation.

Table 2-5 lists the valid ASCII numeric (decimal digit) characters that follow the sign byte.

The address A of the string specifies the first byte (the sign byte) in the string. Bytes that follow in increasing addresses contain digits of decreasing significance.

The length L of the string is in the range of 0 to 31 bytes (0 to 31 digits). The value of a 0 length string is 0 and contains only the sign bit.

Table 2-4 Leading Separate Numeric Sign Characters

Sign	Octal	Decimal	Hex	ASCII Character
+	53	43	2B	+
+	40	32	20	Blank (Space)
–	55	45	2D	–

Table 2-5 ASCII Numeric Characters

ASCII Character (Decimal Digit)	Octal	Decimal	Hex
0	60	48	30
1	61	49	31
2	62	50	32
3	63	51	33
4	64	52	34
5	65	53	35
6	66	54	36
7	67	55	37
8	70	56	38
9	71	57	39

2.1.4.3 Trailing Numeric String Data – Figure 2-8 shows examples of positive and negative values in the trailing numeric string format. Each is specified by the address A of the first byte, which contains the most significant digit, and by the length L of the string in bytes.

All bytes in the string, except the last one, contain one of the ASCII numeric (decimal digit) characters given in Table 2-5.

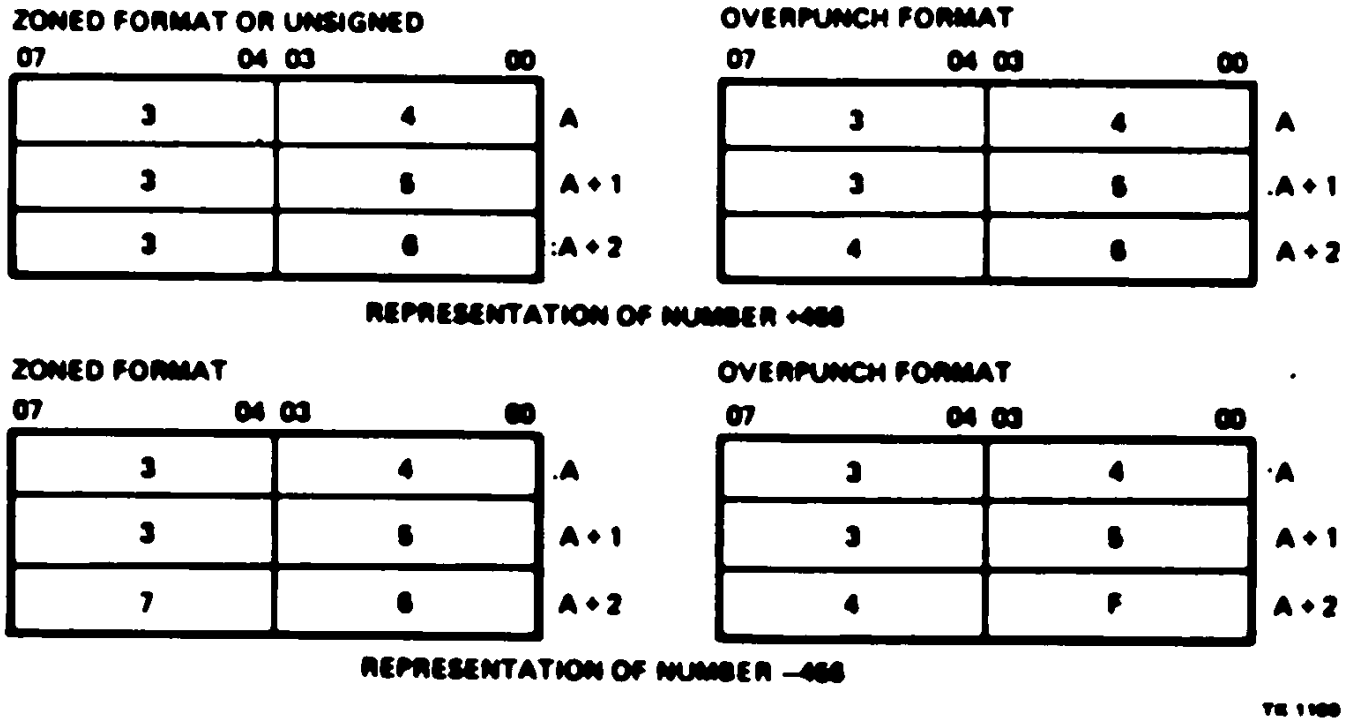


Figure 2-8 Trailing Numeric String Formats

The last byte of the string (in the highest address) is an encoding of the least significant digit and the sign of the string. The VAX numeric string instructions support any type of encoding. However, the following three preferred types of encoding are used by DIGITAL software:

- Unsigned numeric in which there is no sign. The least significant byte contains an ASCII numeric character.
- Zoned numeric.
- Overpunch numeric.

The overpunch format has been used in compilers by manufacturers for many years and various card encodings are still in use. For this reason, several variations in the overpunch format have evolved and are still accepted as input. Table 2-6 shows the valid digit and sign representations for the preferred signed formats.

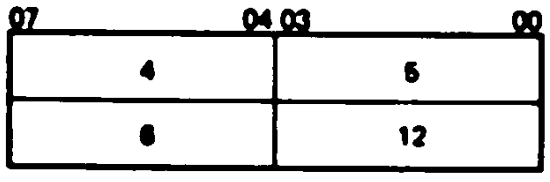
The address A of the string specifies the byte of the string that contain the most significant digit. Increasing byte addresses contain digits of decreasing significance.

The length L of the string is in the range of 0 to 31 bytes (0 to 31 digits). The value of a 0 length string is 0.

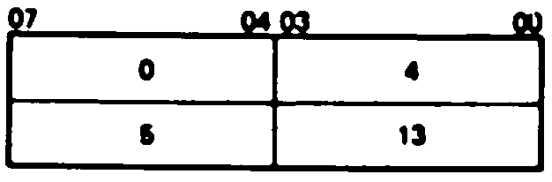
2.1.4.4 Packed Decimal String Data – Figure 2-9 shows two examples of packed decimal values: one with an odd number of digits and one with an even number of digits. For an even number of digits, an extra 0 digit is required in the highest order nibble (bits (07:04) of the first byte in the string).

A packed decimal string is specified by the address A of the first byte of the string and the length L, which is the number of digits in the string. L is not the number of the string in bytes. The bytes are divided into two 4-bit fields (nibbles). Each nibble contains a decimal digit, except for the low nibble. Bits (03:00) of the last byte (the highest address) must contain the sign. The digit and sign representations are shown in Table 2-7.

The preferred sign representation is 12 for "+" and 13 for "–". The length L is the number of digits in the packed decimal string, not including the sign, and is in the range of 0 through 31. Base address A specifies the byte that contains the most significant digit in its high-order nibble. Digits decrease in significance in increasing byte addresses and from the high to low nibble in a byte.



REPRESENTATION OF VALUE (+466) WITH ODD NUMBER OF DIGITS



REPRESENTATION OF VALUE (–46) WITH EVEN NUMBER OF DIGITS

TK 1174

Figure 2-9 Packed Decimal String Format

Table 2-6 Trailing Numeric Digit and Sign Representations

Decimal Digit	Zoned Numeric Format			ASCII Characters	Overpunch Format			ASCII Character	
	Octal	Decimal	Hex		Octal	Decimal	Hex	Normal	Alternate
+0	60	48	30	0	173	123	7B	{	{ ?
+1	61	49	31	1	101	65	41	A	a
+2	62	50	32	2	102	66	42	B	b
+3	63	51	33	3	103	67	43	C	c
+4	64	52	34	4	104	68	44	D	d
+5	65	53	35	5	105	69	45	E	e
+6	66	54	36	6	106	70	46	F	f
+7	67	55	37	7	107	71	47	G	g
+8	70	56	38	8	110	72	48	H	h
+9	71	57	39	9	111	73	49	I	i
–0	160	112	70	p	175	125	7D	}	] '
–1	161	113	71	q	112	74	4A	J	j
–2	162	114	72	r	113	75	4B	K	k
–3	163	115	73	s	114	76	4C	L	l
–4	164	116	74	t	115	77	4D	M	m
–5	165	117	75	u	116	78	4E	N	n
–6	166	118	76	v	117	79	4F	O	o
–7	167	119	77	w	120	80	50	P	p
–8	170	120	78	x	121	81	51	Q	q
–9	171	121	79	y	122	82	52	R	r

Table 2-7 Packed Decimal Digit and Sign Representations

Digit or Sign	Octal	Decimal	Hex
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7
8	10	8	8
9	11	9	9
+	12,14,16,17	10,12,14,15	A,C,E,F
−	13,15	11,13	B,D

2.1.4.5 Variable-Length Bit Field Data – Figure 2-10 shows the variable-length bit field format. The bit field occupies the shaded area.

A variable-length bit field may occupy from 0 to 32 contiguous bits and may be arbitrarily located with respect to byte boundaries. It is specified by the base address A, the starting bit position P (positive or negative with respect to bit (00) of the byte at address A), and the field size S in bits.

The bit field instructions interpret the field as a signed or unsigned integer. As an unsigned integer, the bit values increase in significance from 0 through S-1. As a signed integer, it is processed in two's complement form with bit S-1 as the sign bit.

The variable-length bit field format is used to store small integers packed together in a larger data structure. This saves memory space when many small integers are part of a larger structure. A specific case is that of one bit where this form efficiently stores and accesses individual flags.

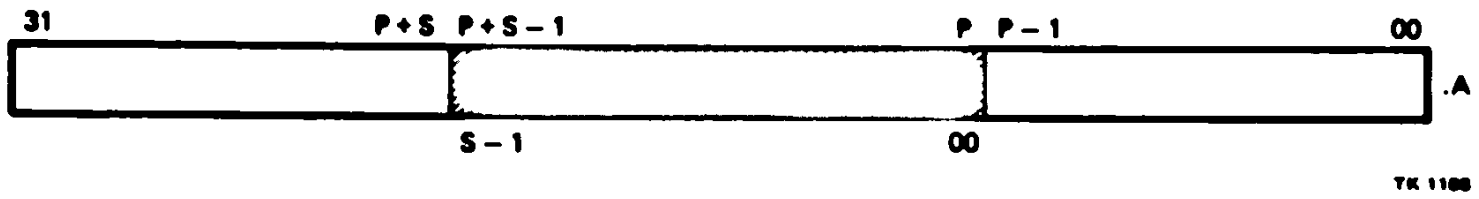


Figure 2-10 Variable Length Bit Field Format

2.1.4.6 Queue Data – Queue data contains lists or tables. Queues consists of circular entries that are forward- and backward-linked by headers each of which consists of two longwords. The first longword of a header specifies the next queue entry; the second specifies the preceeding entry. VAX supports the following types of queue entries:

- Absolute — The header contains the absolute address of the last entry and next entry as shown in Figure 2-11.
- Self-relative — The header contains addresses that are displacements from the current entry and are used to select the last entry and next entry as shown in Figure 2-12.

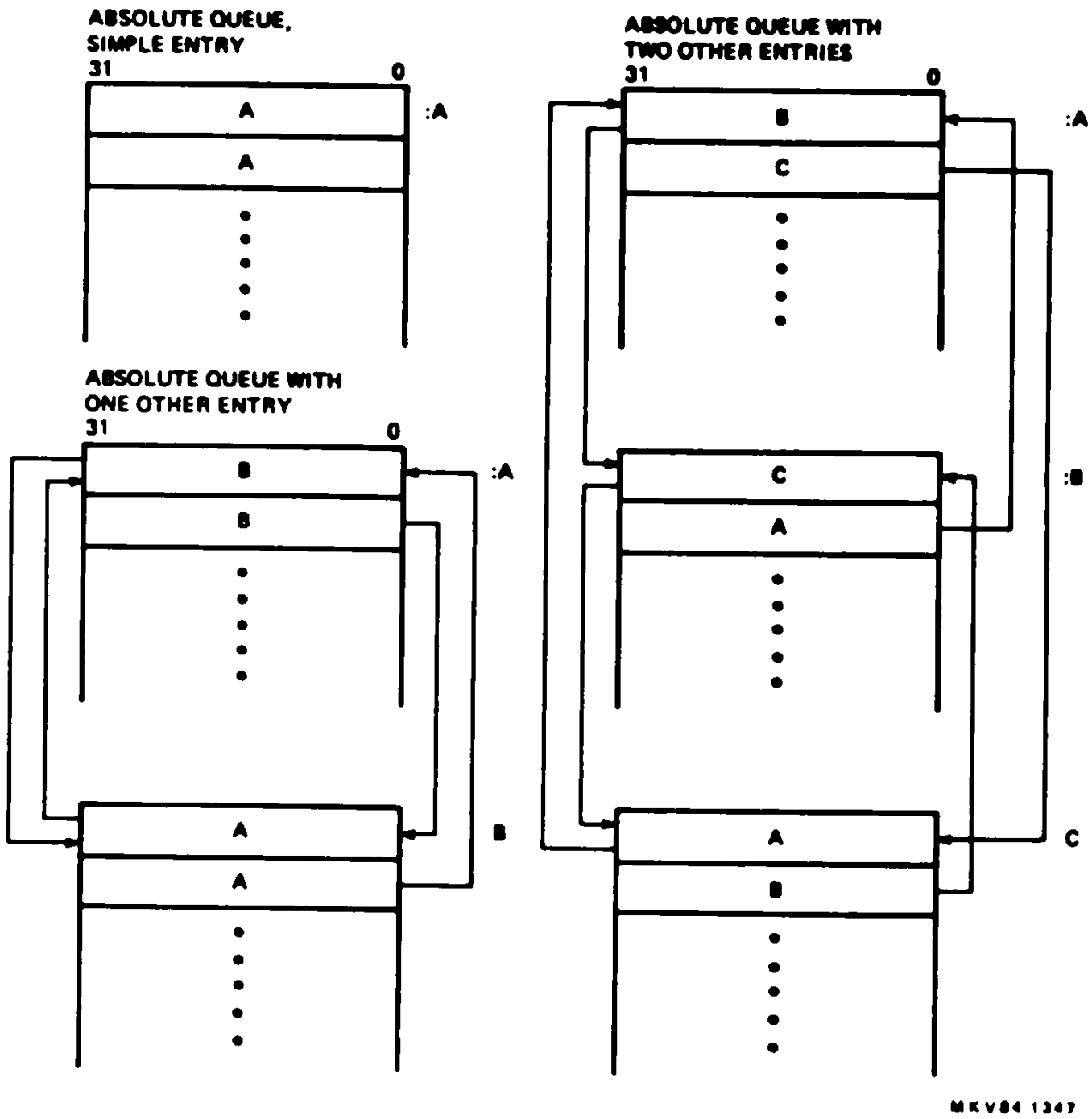


Figure 2-11 Absolute Queue Entry Formats

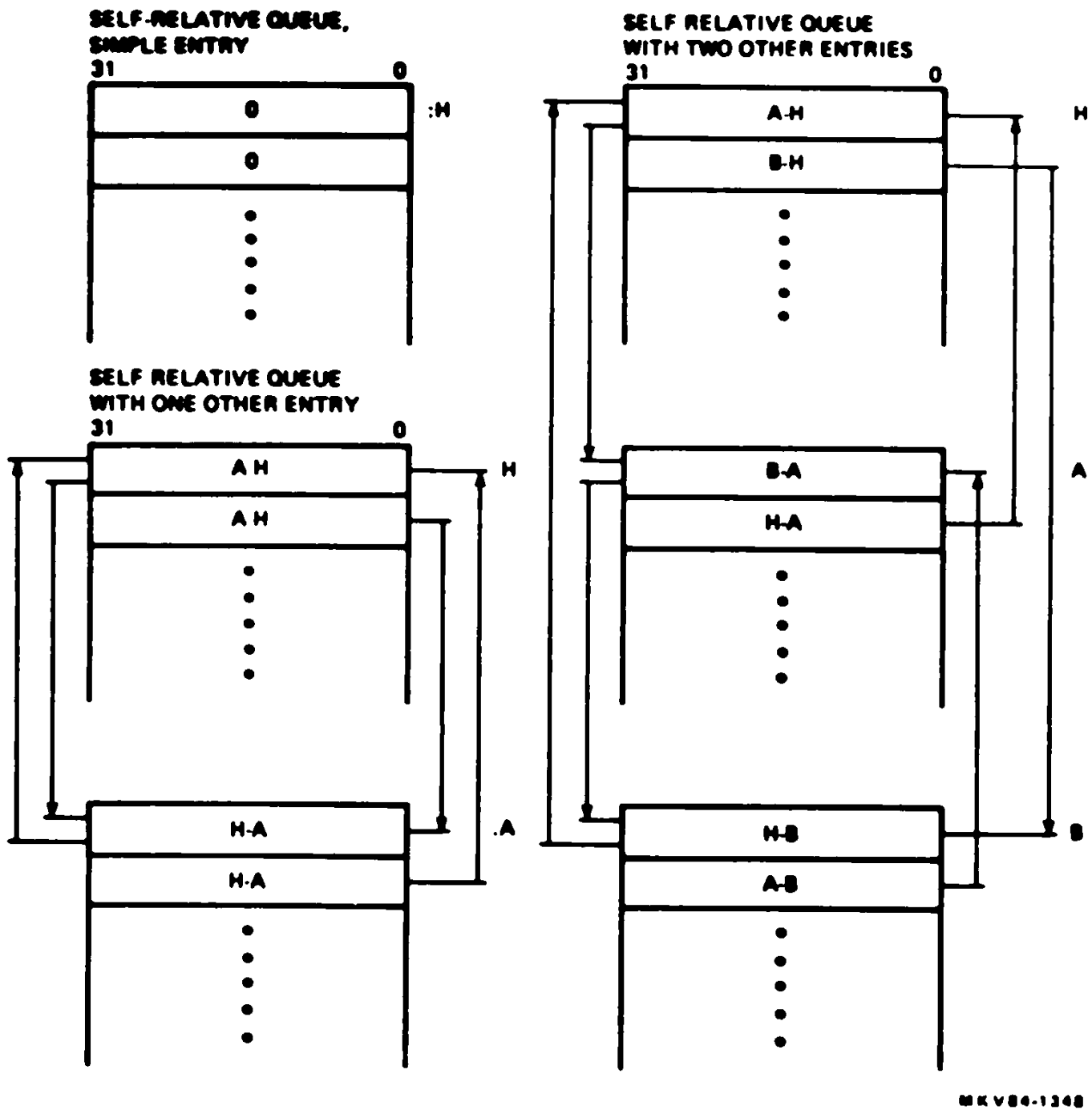


Figure 2-12 Self-Relative Queue Entry Formats

2.1.5 System Registers

VAX architecture provides hardware and software registers that are commonly used by all processes. While some registers are available for general use, others are reserved for more privileged functions.

2.1.5.1 General Registers – Sixteen general purpose hardware registers are used for the temporary storage of data or addresses. A register is accessed when the register number is expressed in an operand specifier or is selected by a machine operation. With certain limitations, the general registers are available for the uses described in Table 2-8.

2.1.5.2 Stacks – Stacks (also called pushdown lists or last-in/first-out queues) are maintained in memory by the operating system for the following functions:

- To create temporary storage space
- For the nesting of recursive routines

- To save general registers, including the PC, when entering a subroutine and to restore them when exiting the subroutine
- To save PC, PSL, and general register contents on an interrupt or exception and during context switches

Table 2-8 General Register Description

Register	Description
R0–R5	Registers R0 through R5 are generally available for any use, but are loaded with specific values by instructions that may be interrupted during their execution
R6 R11	Registers R6 through R11 have no reserved functions for the hardware or operating system and the user software may assign specific uses
R12*	*Argument Pointer (AP) Register* The VAX-11 procedure call convention uses a data structure, called an argument list, that uses the AP as the base address The CALL instructions load the AP in accordance with this convention, but there is no hardware or software restriction on the use of the register for other purposes
R13*	*Frame Pointer (FP) Register* – The VAX-11 procedure call convention builds a data structure based on the kernel stack called a stack frame The CALL instructions load the FP with the base address of the stack frame The return (RET) instruction then depends on the FP for the base address of the stack frame. VAX-11 software also depends on the FP for the correct reporting of certain exception conditions.
R14	*Stack Pointer (SP) Register* – Most software assumes that the SP points to the kernel stack for the current process Except for the PC, there is no restriction on the use of other registers as stack pointers However, instructions that use the kernel stack, without identifying a stack pointer in the operand specifier, always select the SP.
R15	*Program Counter (PC) Register* – The PC points to the next byte of the program It is continually updated as the program progresses and may only be altered by the branch and jump instructions It must not be used for any other purpose

* VAX software uses two types of subprograms subroutines and procedures While registers R12 and R13 may be used for any purpose, they have reserved functions when the procedure CALL and RET instructions are used

A stack is defined by a block of memory and a stack pointer register that points to the top of the stack The top of the stack is the current location that is read when an item is added (pushed) to or is removed (popped) from the stack.

An item is pushed on a stack by first decrementing the stack pointer by the length of the item, then storing the item at the address selected by the updated stack pointer (The length of the item then fills the area in ascending locations) When an item is popped off the stack, the length of the item is added to the stack pointer after the operation.

Many processor operations make use of the kernel stack without identifying the stack pointer in an operand specifier. This occurs in the instructions used in calling and returning from subroutines, and in processor sequences that initiate and terminate the interrupt or exception service routines. In such cases, the processor always uses the stack addressed by the SP register (R14).

Exceptions, interrupts, and system services are not performed on the same stack as used by the user mode programs. The processor maintains five other internal registers as pointers to separate blocks of memory for use as stacks, and uses one or another as the SP depending on the current access mode and the interrupt stack bit in the processor status longword.

Whenever the current access mode or interrupt stack bits change, the processor saves the contents of the SP in the internal register selected by the old value of those bits. It then loads the SP from the register selected by the new value. There is one interrupt stack for the entire system, but the kernel, executive, supervisor, and user mode stacks change for each process on the system.

Figure 2-13 shows the relationship between the five stacks and the multiple processes. The multiple-stack mechanism provides the following advantages over a single stack:

- User mode programs are not subject to sudden and nonreproducible changes in the data beyond the end of their stack.

- The integrity of a privileged mode program cannot be compromised by a less privileged caller. The privileged code is not in danger of running out of space, even if the caller has completely filled its own stack, because separate blocks of memory are allocated to the stack associated with each mode.

- Privileged mode programs are not vulnerable to accidental destruction of the stack pointer by less privileged programs.

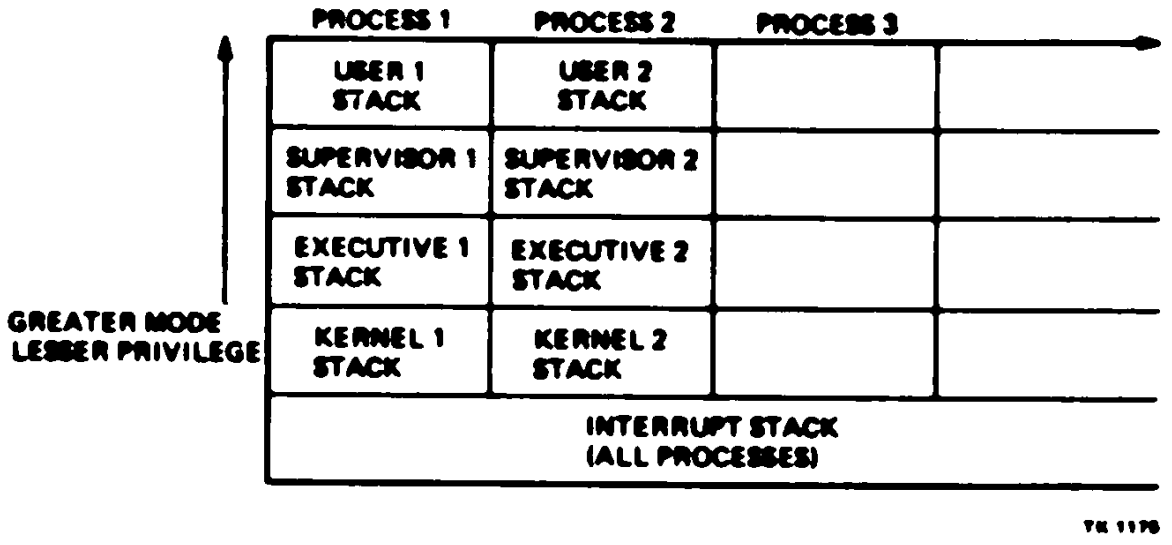


Figure 2-13 Stack and Process Relationships

2.1.5.3 Processor Registers – Processor registers include privileged memory management registers and registers related to system control and operation. These registers identify the currently executing process for the operating system and are accessed by privileged instructions that can only be issued in the kernel mode.

Table 2-9 lists the processor registers in their hexadecimal (hex) address order. Refer to Chapter 5, *VAX Maintenance Handbook for the VAX-11/780* (EK-VAXV2-HB) for figures that show register bit usages.

Table 2-9 Processor Registers

Register Number (Hex)	Mnemonic	Name
00	KSP	Kernel stack pointer
01	ESP	Executive stack pointer
02	SSP	Supervisor stack pointer
03	USP	User stack pointer
04	ISP	Interrupt stack pointer
05–07	Reserved	
08	P0BR	P0 base register
09	P0LR	P0 length register
0A	P1BR	P1 base register
0B	P1LR	P1 length register
0C	SBR	System base register
0D	SLR	System length register
0E–0F	Reserved	
10	PCBB	Process control block base
11	SCBB	System control block base
12	IPL	Interrupt priority level
13	ASTR	Asynchronous trap level
14	SIRR	Software interrupt request
15	SISR	Software interrupt summary
16–17	Reserved	
18	ICCS	Interval clock control/status
19	NICR	Next interval count
1A	ICR	Interval count
1B	TODR	Time of day
1C–1F	Reserved	
20	RXCS	Console receive control/status
21	RXDB	Console receive data buffer
22	TXCS	Console transmit control/status
23	TXDB	Console transmit data buffer
24–27	Reserved	
28	ACCS	Accelerator control/status
29	ACCR	Accelerator reserved
2A–2B	Reserved	
2C	WCSA	WCS address
2D	WCSD	WCS data
2E–2F	Reserved	

Table 2-9 Processor Registers (Cont)

Register Number (Hex)	Mnemonic	Name
30	SBIFS	SBI fault/status
31	SBIS	SBI silo
32	SBISC	SBI silo comparator
33	SBIMT	SBI maintenance
34	SBIER	SBI error
35	SBITA	SBI timeout address
36	SBIQC	SBI quadword clear
37	Reserved	
38	MME	Memory management enable
39	TBIA	TB invalidate all
3A	TBIS	TB invalidate single
3B	Reserved	
3C	MBRK	Microprogram breakpoint
3D	PMR	Performance monitor
3E	SID	System identification
3F	Reserved	

2.1.5.4 Processor Status Longword (PSL) – The 32-bit PSL format is shown in Figure 2-14. PSL monitors a number of processor states and variables that are associated with each process. Bits (31:16) contain privileged processor status information. Bits (15:00) are the processor status word (PSW). The PSW contains nonprivileged information and the defined bits may be controlled by any program. Any program can perform the return from exception or interrupt (REI) instruction which affects the PSL. However, no bits are altered that would allow a process to increase its privilege or create an undefined state in the processor.

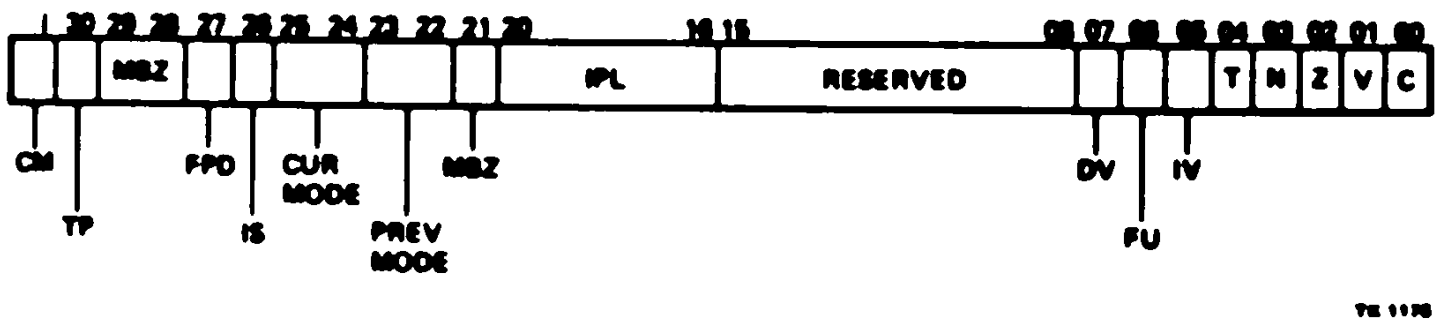


Figure 2-14 Processor Status Longword (PSL)

Table 2-10 provides brief general descriptions of the PSL status bits and flags. Bits (07:04) are the trap enable bits. Two more trap conditions, division-by-zero and floating overflow, are always enabled. Bits (03:00), the condition codes, reflect the resultant status of the most recently executed instruction.

Refer to the *VAX Architecture Handbook* (EB-19580) for details of how machine instructions and processor states affect the PSL.

Table 2-10 Processor Status Longword (PSL) Description

Bit	Name	Description
(31)	Compatibility Mode (CM)	When CM is clear, the processor is in the native mode and executes VAX 11 instructions When it is set, the processor is in the PDP-11 compatibility mode and executes PDP-11 instructions
(30)	Trace Pending (TP)	TP is used by the processor to ensure that only one trace trap occurs during each instruction performed with the trace bit (bit (04)) set
(29·28)		Reserved and must be zero (MBZ)
(27)	First Part Done (FPD) Flag	FPD is used by the processor in instructions that may be interrupted or page faulted during their execution If FPD is set, when the processor returns from an exception or interrupt, the instruction resumes from where it left off rather than restarting
(26)	Interrupt Stack (IS) Flag	When set, IS indicates that the processor is using the common interrupt stack rather than one of four stacks associated with the current mode When IS is set, the current mode is always kernel and the software operating on the interrupt stack has full kernel mode privileges
(25·24)	Current Mode (CM) Field	The CM field selects the processor mode that determines the access privilege of the currently executing program. The CM field values are as follows 0 - Kernel — Most privileged 1 - Executive 2 - Supervisor 3 - User — Least privileged The processor uses the CM field to grant access privileges in the following ways Privileged instructions are not executed unless the processor is in kernel mode The memory management logic controls access to virtual addresses depending on the program's current access mode, the type of reference(read or write), and the protection code assigned to each page of the address space.

Table 2-10 Processor Status Longword (PSL) Description (Cont)

Bit	Name	Description
⟨23:22⟩	Previous Mode (PM) Field	The PM field stores the value of the CM field for the most recent exception that changed it to the present mode from a less privileged one PM is used by the PROBE instruction in privileged routines to determine if a caller at the previous mode has sufficient privileges to reference a given area of memory
⟨21⟩		Reserved and must be zero (MBZ)
⟨20:16⟩	Interrupt Priority Level (IPL)	The IPL selects the current interrupt priority level of the processor In order for an interrupt to be acknowledged by the processor, it must be at a higher priority than the current IPL All software runs at IPL 0 for the processor to acknowledge and service interrupt requests at any priority level The interrupt service routine for each request, however, sets its IPL to that of the honored request, blocking all requests of equal or lower priority The 31 priority levels above zero are numbered in 01 through 1F. Interrupt levels 01 through 0F are software generated requests Levels 10 through 17 are reserved for use by hardware peripheral devices and controllers Levels 18 through 1F are used for urgent conditions such as the interval time clock, fatal errors and power failures
⟨15:08⟩		Reserved and must be zero (MBZ)
⟨07⟩	Decimal Overflow (DV) Trap Enable	When set, DV causes a decimal overflow trap after an instruction that produces a decimal result whose absolute value is too large to be represented in the destination space The stored result then consists of the low-order digits and sign of the algebraically correct result
⟨06⟩	Floating Underflow (FU) Trap Enable	When set, FU causes a floating underflow trap after an instruction that produces a floating result whose absolute value is too small to be represented When a floating underflow occurs, it stores a zero result regardless of the state of FU.

Table 2-10 Processor Status Longword (PSL) Description (Cont)

Bit	Name	Description
⟨05⟩	Integer Overflow (IV) Trap Enable	When set, IV causes an integer overflow trap after any instruction that produces an integer result that cannot be correctly represented in the space provided. (Bit ⟨01⟩, the V condition code, is not dependent on the state of IV)
⟨04⟩	Trace (T) Enable	When set, T causes a trace trap to occur after the execution of the next instruction This facility is used during software debugging and performance analysis to step through a program one instruction at a time.
⟨03⟩	Negative (N) Sign Bit	N is set by an instruction in which the stored result is negative (less than zero) It is cleared by an instruction where the result is positive (or zero). For instructions that affect N according to a stored result, it reflects the actual sign result, even if it is algebraically incorrect because of an overflow
⟨02⟩	Zero (Z) Bit	Z is set by an instruction that stores a result that is exactly zero and is cleared if the result is not zero. Z reflects the actual result, even on an overflow
⟨01⟩	Overflow (V) Bit	V is set after an arithmetic operation if the magnitude of the algebraically correct result is too large to be represented in the available space It is cleared if the result fits Instructions in which an overflow is impossible or meaning-less either clear V or leave it unaffected Overflow conditions that set V also cause traps if the appropriate trap enable bits are set
⟨00⟩	Carry (C) Bit	C is set after an arithmetic operation causes the most significant bit to generate a carry or a borrow It is cleared after an operation that does not generate a carry or borrow It is either cleared or unaffected by other instructions The C bit determines the operation of conditional branch instructions It serves as an input variable to the add with carry (ADWC) and subtract with carry (SBWC) instructions

2.2 NATIVE MODE INSTRUCTION FORMATS

Figure 2-15 shows the general format for VAX native mode instructions. Native mode instructions vary in length and format depending on the instruction class and addressing type. The operation code (opcode) occupies the first byte position in the instruction stream (I-stream) and the instruction executes in ascending byte addresses. (Compatibility mode instructions follow the standard PDP-11 16-bit format and are stored as two contiguous bytes in memory.)

OPERAND SPECIFIER N (1 OR 2 BYTES)
IMMEDIATE DATA (1, 2, 4, OR 8 BYTES)
OPERAND SPECIFIER 2 (1 OR 2 BYTES)
SPECIFIER EXTENSION (1 TO 6 BYTES)
OPERAND SPECIFIER 1 (1 OR 2 BYTES)
OPCODE (1 OR 2 BYTES)

Figure 2-15 VAX-11 Native Mode Instruction Format

2.2.1 Operation Code (Opcode)

Native mode instructions use a one-byte opcode that specifies the operation to be performed. A separate set of two-byte opcodes performs extended precision arithmetic operations. Refer to the *VAX Architecture Handbook* (EB-19580) for a list of the native instruction opcodes.

A native mode instruction may only consist of the opcode or it may contain the opcode and one or more operand specifiers. The opcode selects the number of operands in the operation and specifies one of the following data types (described in Section 2.1.1.3):

- Byte
- Word
- Longword
- Quadword
- Octaword
- F_floating
- D_floating
- G_floating
- H_floating

2.2.2 Operand Specifier (Opspec)

An operand specifier provides a method of access to the operand. It either specifies or calculates the operand address by supplying the addressing mode plus memory address or register select information.

2.2.2.1 Addressing Types – Each operand specifier selects one of four addressing types in its format. The branch instructions use only one type. Instructions that perform logical or arithmetic functions select either the general registers or the program counter. The four types of addressing are shown in Table 2-11.

Table 2-11 Summary of Addressing Types

Addressing Type	Description
Branch Addressing	The branch, jump, and case instructions load the PC to relocate the program to other areas of memory.
General Register Addressing	Instructions perform logical or arithmetic operations on general register contents. They also perform operations on data in memory using the register contents as virtual addresses.
Program Counter Addressing	Instructions perform operations on virtual memory addresses referenced from the current contents of the PC. The address or offset for the operand is located in the instruction stream immediately following the operand specifier. In the immediate mode, the operand immediately follows the operand specifier. In the literal mode, the operand specifier includes the operand value as an integral part of the format.
Index	The index modes expand the normal addressing modes by using two registers. One stores the base address of a list or table. The second stores an index value that is used to calculate a specific address or entry in the table. The PC is used in formats that provide the base address or displacement in the instruction stream itself.

2.2.2.2 Access Types – The operand specifier first determines the address of the operand. The instruction then processes the contents of the address as a source or destination operand under one of the access types shown in Table 2-12.

Table 2-12 Summary of Access Types

Access Type	Description
Read	Accesses the operand as read only.
Write	Accesses the operand as write only.
Modify	Reads then writes the operand, with or without modification.
Address	Does not directly access the operand. Address calculation takes place until the actual operand address is obtained. (The data type specified by the opcode indicates the operand size and is used in calculating the address.)
Variable Bit Field	If only a general register is specified, the bit field is in the general register. Otherwise, address calculation takes place until the actual byte address of the operand is obtained. The field position (bit offset) is then added to this base address and the operand is determined from its size.
Branch	Loads the program counter (does not access an operand). The operand specifier may contain an absolute address or a branch displacement. The data type (byte or word) indicates the size of any branch displacement.

2.3 NATIVE MODE ADDRESSING

This is a description of the addressing types used by the operand specifiers in VAX native mode instructions.

2.3.1 Branch Addressing

By loading the program counter (R15), the branch and jump instructions provide a method of relocating program control to other areas of memory. Branch instructions use two address sizes, byte or word, in the displacement format shown in Figure 2-16. Jump instructions, however, use other PC addressing modes, allowing relocation to a wider range of addresses. (Descriptions of all branch and jump instruction forms are provided in Sections 2.4.7 through 2.4.9.)

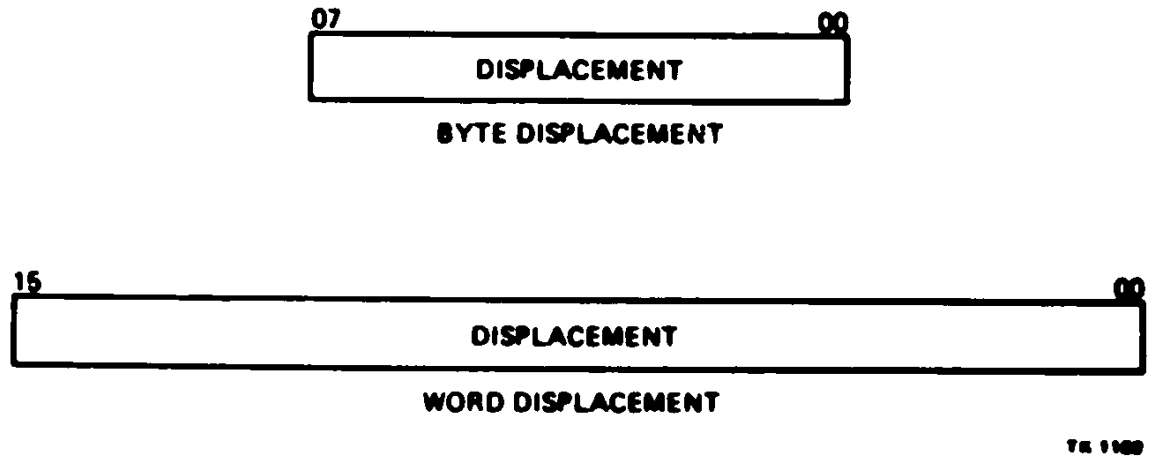


Figure 2-16 Branch Addressing Operand Qualifier

2.3.2 General Register Addressing

This addressing type selects one of the fifteen general registers, R0 through R14, in the operand specifier. General registers may be used as simple buffers to temporarily store data or as accumulators to perform logical and arithmetic operations. They may also contain the virtual memory address of the data or operand.

When data is loaded to a general register, it is stored in the same format as in memory. If a quadword or floating datum is loaded, it is stored in two adjacent registers.

The use of R15, the program counter (PC), is not allowed. For this reason, also, the stack pointer (SP) cannot be specified for an operand that needs two adjacent registers.

If the selected register is used as a pointer, the register contains the virtual address of the operand, rather than the operand itself. The register is used as a base register if it contains the base address of a data structure, such as a table or a queue string. It may also be used as a pointer that steps forward (autoincrement addressing) or backward (autodecrement addressing) through consecutive memory locations.

Index addressing is useful for processing tabular data or manipulating stacks. When a register is used as an index register, it generates an offset, according to the data size, that is added to the base address of the operand.

Figure 2-17 shows the operand specifier format for the general register addressing modes. Bits ⟨03:00⟩ select a general register R0 through R14, and bits ⟨07:04⟩ specify one of the modes shown in Table 2-13.

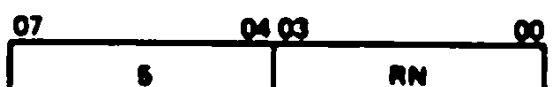


Figure 2-17 General Register Mode Operand Specifier

Table 2-13 General Register Addressing Modes

Bits ⟨07:04⟩ Value (Hex)	General Register Addressing Mode
5	Register
6	Register deferred
8	Autoincrement
9	Autoincrement deferred
7	Autodecrement
A	Displacement (byte)
C	Displacement (word)
E	Displacement (longword)
B	Displacement deferred (byte)
D	Displacement deferred (word)
F	Displacement deferred (longword)
4	Index modes

2.3.2.1 Register Mode – Register mode moves data to or from a general register or performs logical or arithmetic operations on the register contents. Because they are hardware registers within the CPU, the general registers provide speed advantages when used in operations on frequently accessed variables.

2.3.2.2 Register Deferred Mode – For register deferred mode, the selected register contains the address of the operand rather than the operand itself. This provides one level of indirect addressing over the register mode. The deferred modes are useful when the address of an operand must be calculated.

2.3.2.3 Autoincrement Mode – In autoincrement mode, the contents of the selected register (Rn) are first used as the address of the operand. The register is then incremented as follows. The size of the operand (in bytes) is added to the contents of register Rn: 1 for byte; 2 for word; 4 for longword or F_floating; 8 for quadword,D_floating or G_floating; or 16 for octaword or H_floating. The contents of Rn are replaced with the result and are used as the address of the next operand.

This mode is useful for stack and array processing and allows stepping of a pointer through sequential elements in a table of operands. It assumes that the register contents are the address of the operand and increments them to provide the address of the next sequential location. Although most useful for handling tables, this is a general mode and may be used for a variety of purposes.

2.3.2.4 Autoincrement Deferred Mode – In autoincrement deferred mode, the selected register (Rn) contains a longword address that points to the address of the operand. After the operand address has been retrieved, 4 is added to the contents of register Rn. The contents of Rn are replaced with the result and the updated contents are used as the address of the address of the next operand. (A quantity of 4 is used because four bytes are stored in the longword address held by Rn.)

2.3.2.5 Autodecrement Mode – In autodecrement mode, the contents of the selected register (Rn) are first decremented and then used as the address of the operand. The size of the operand (in bytes) is first subtracted from the contents of register Rn: 1 for byte; 2 for word; 4 for longword or F_floating; 8 for quadword D_floating or G_floating; or 16 for octaword or H_floating. The contents of Rn are replaced with the result and used as the address of the operand.

2.3.2.6 Displacement Mode – In displacement mode addressing (Figure 2-18), if the displacement value is a byte or a word, it is first sign-extended to 32 bits. The displacement value is added to the contents of the selected register (a base address) and the result is used as the operand address. (This is the equivalent of the index mode in PDP-11 addressing.)

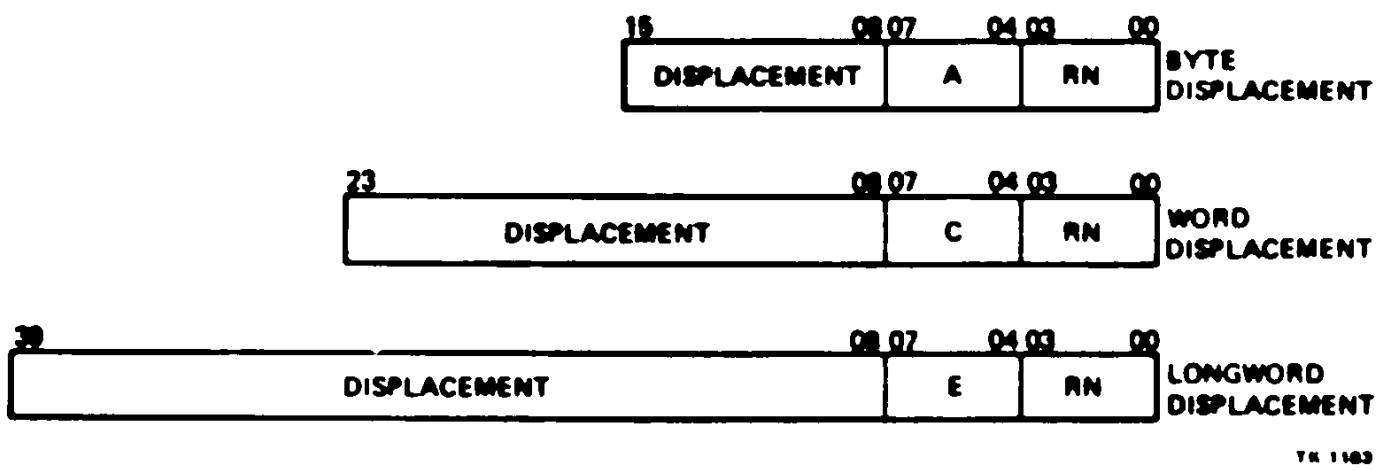


Figure 2-18 Displacement Mode Operand Specifier

VAX architecture provides for 8-bit, 16-bit or 32-bit offsets because most program references occur within small discrete portions of the address space. A 32-bit offset is not always necessary and the 8- and 16-bit offsets result in large savings in memory space by using fewer bits.

2.3.2.7 Displacement Deferred Mode – In displacement deferred mode (Figure 2-19), if a byte or word, the displacement value is first sign-extended to 32 bits. The displacement value is added to the contents of the selected register (a base address). The result is used as the longword address of the address of the operand.

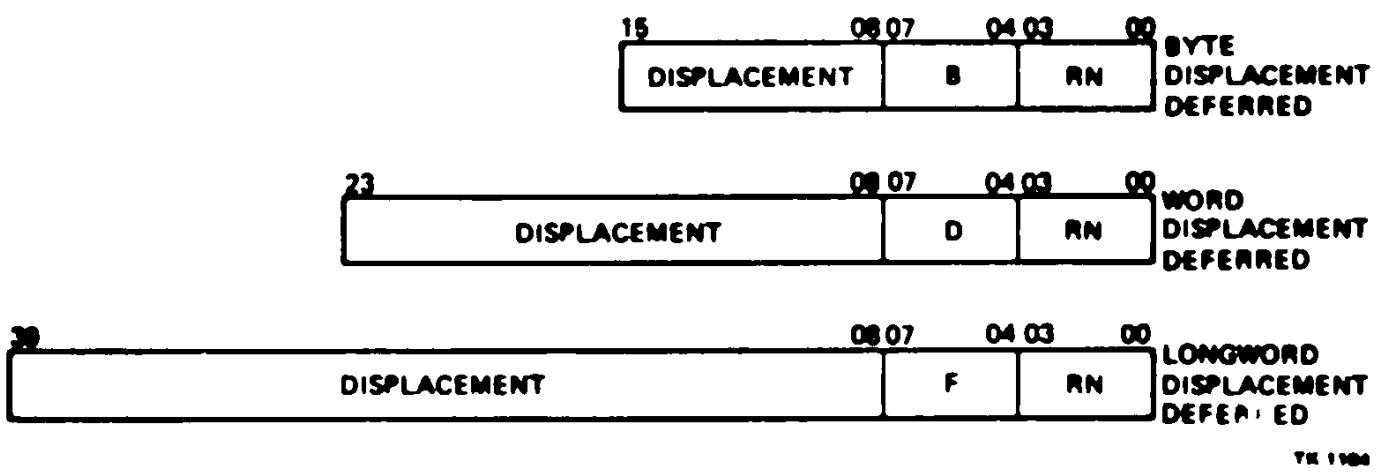


Figure 2-19 Displacement Deferred Mode Operand Specifier

2.3.2.8 Summary of General Register Addressing Modes – Table 2-14 summarizes the general register addressing modes and includes the following information: the mode field value for each operand specifier, the assembler notation, the indexable modes, the operand access types for each mode. It also shows the results of using the stack pointer (R14) or program counter (R15) in each mode.

2.3.3 Program Counter Addressing

The program counter addressing modes make their references from the contents of the PC (R15) which is incremented as the instruction is evaluated and executed.

The program counter addressing modes also use the operand specifier format shown in Figure 2-17. However, bits (03:00) hold a value of F (hex) and bits (07:04) select one of the modes shown in Table 2-15.

The execution of these modes is the same as for the general register modes. However, the program counter contents are used instead of a general register. The following paragraphs provide a brief description of the program counter addressing modes and operand specifier formats.

Table 2-14 Summary of General Register Addressing Modes

Mode Hex	Field Dec	Name	Assembler	Access r	m	w	a	v	PC	SP	Indexable?
4	4	indexed	i[Rx]	y	y	y	y	y	f	y	f
5	5	register	Rn	y	y	y	f	y	u	uq	f
6	6	register deferred	(Rn)	y	y	y	y	y	u	y	y
7	7	autodecrement	-(Rn)	y	y	y	y	y	u	y	ux
8	8	autoincrement	(Rn)+	y	y	y	y	y	p	y	ux
9	9	autoincrement deferred	@(R)+	y	y	y	y	y	p	y	ux
A	10	byte displacement	B^D (Rn)	y	y	y	y	y	p	y	y
B	11	byte displacement deferred	@B^D (Rn)	y	y	y	y	y	p	y	y
C	12	word displacement	W^D (Rn)	y	y	y	y	y	p	y	y
D	13	word displacement deferred	@W^D (Rn)	y	y	y	y	y	p	y	y
E	14	longword displacement	L^D (Rn)	y	y	y	y	y	p	y	y
F	15	longword displacement deferred	@L^D (Rn)	y	y	y	y	y	p	y	y

Legend
- i - any indexable addressing mode
- f - reserved addressing mode fault
- p - Program counter addressing
- u - Unpredictable
- uq - Unpredictable for quadword, octaword, D_floating, G_floating, and H_floating (and field if position + size is greater than 32)
- ux - Unpredictable if index register same as base register
- y - yes, always valid addressing mode
- r - read access
- m - modify access
- w - write access
- a - address access
- v - variable bit field access

Table 2-15 Program Counter Addressing Modes

Bits (07:04) Value (Hex)	Program Counter Addressing Mode
8	Immediate
9	Absolute
A	Relative (byte)
C	Relative (word)
E	Relative (longword)
B	Relative deferred (byte)
D	Relative deferred (word)
F	Relative deferred (longword)
0-3	Literal

2.3.3.1 Immediate Mode – The location immediately following the operand specifier contains the operand constant (Figure 2-20).

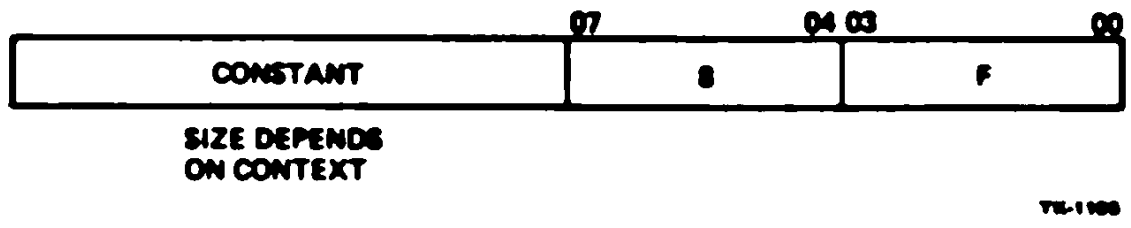


Figure 2-20 Immediate Mode Operand Specifier

2.3.3.2 Absolute Mode – The contents of the location that follows the operand specifier are used as the operand address (Figure 2-21). This is an absolute address that remains constant regardless of the memory area in which the assembled instruction is executed.

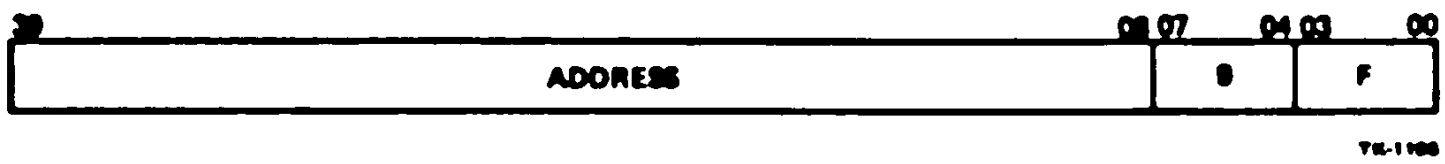


Figure 2-21 Absolute Mode Operand Specifier

2.3.3.3 Relative Mode – The displacement value that follows the operand specifier is added to the PC (Figure 2-22). The result is the address of the operand. This mode is useful for writing position independent code because the referenced location is always fixed, relative to the PC.

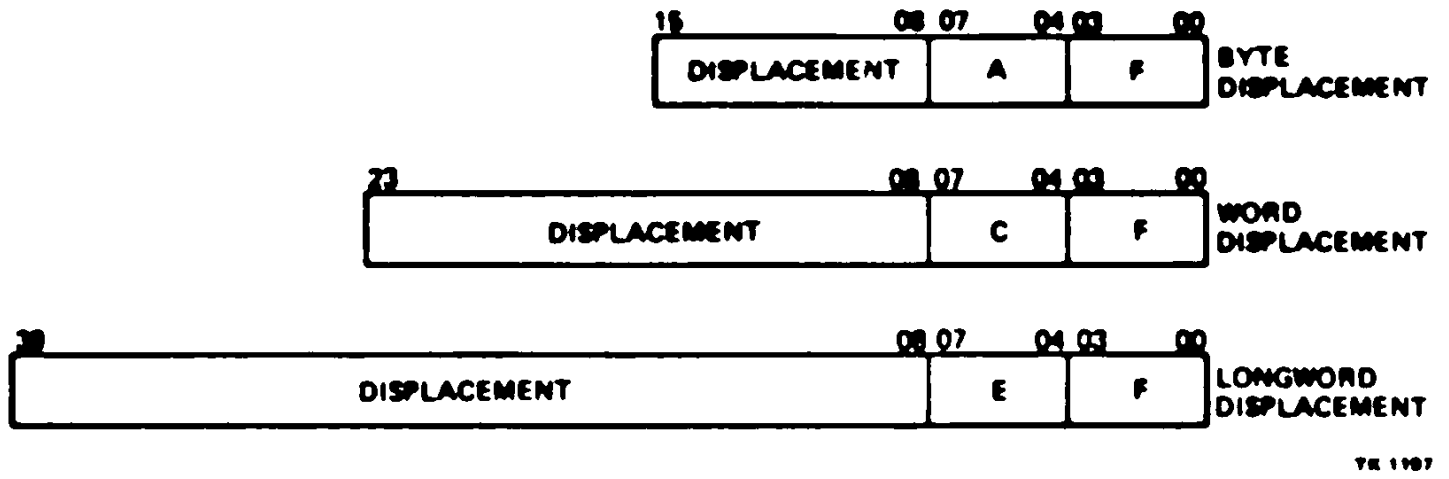


Figure 2-22 Relative Mode Operand Specifier

2.3.3.4 Relative Deferred Mode – The displacement value that follows the operand specifier is added to the PC (Figure 2-23). The result is the longword address of the address of the operand. This mode is useful for processing tables of addresses.

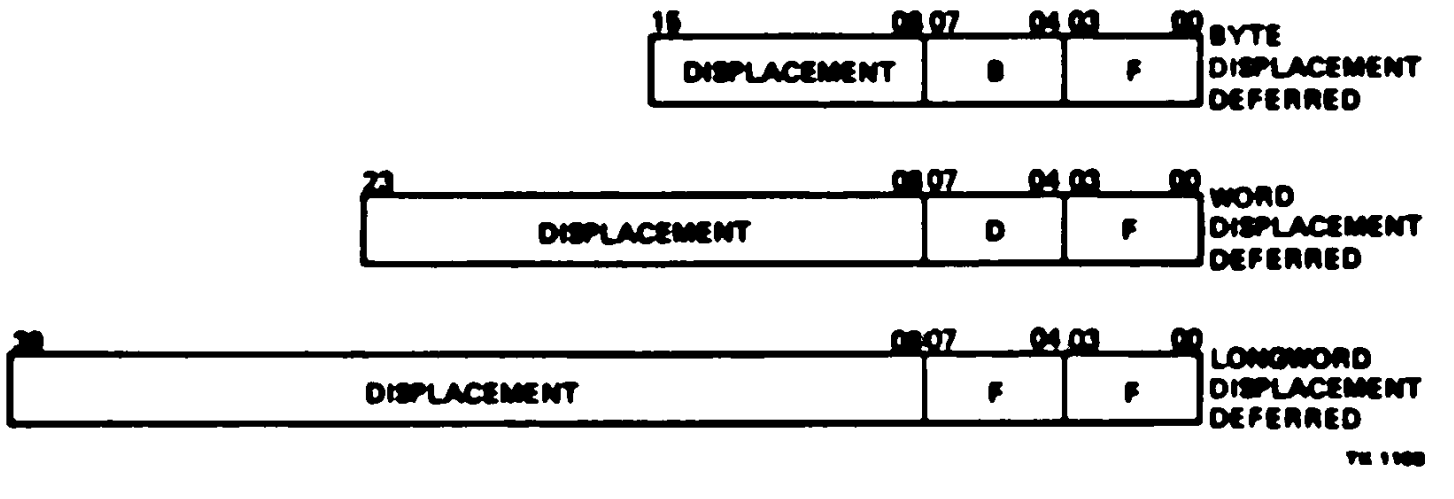


Figure 2-23 Relative Deferred Mode Operand Specifier

2.3.3.5 Literal Mode – Literal mode (Figure 2-24) may be used in place of the immediate mode as an efficient way of specifying integer constants in the range of 0 to 63 (decimal). Values in that range are called short literals. Long literals (values above 63) can be expressed using the immediate mode. Bits ⟨07:06⟩ of the mode specifier are always zero. The value of bits ⟨07:04⟩ will be 0, 1, 2, or 3 depending on the short literal value in bits ⟨05:00⟩.

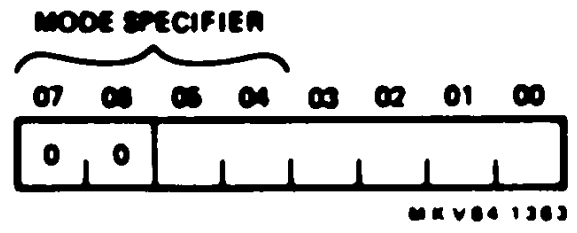


Figure 2-24 Literal Mode
Operand Specifier

In addition to six-bit integers, the literal mode can also be used to express floating point values. As a floating point operand (Figure 2-25), the floating literal is composed of a three-bit exponent (EXP) field and a three-bit fraction (FRAC) field.

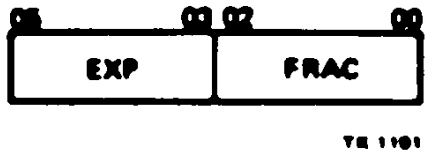


Figure 2-25 Floating Literal Format

Table 2-16 shows all possible floating literal values that can be expressed in the operand specifier.

Table 2-16 Floating Literals

Exp	Fraction 0	1	2	3	4	5	6	7
0	1/2	9/16	5/8	11/16	3/4	13/16	7/8	15/16
1	1	1 1/8	1 1/4	1 3/8	1 1/2	1 5/8	1 3/4	1 7/8
2	2	2 1/4	2 1/2	2 3/4	3	3 1/4	3 1/2	3 3/4
3	4	4 1/2	5	5 1/2	6	6 1/2	7	7 1/2
4	8	9	10	11	12	13	14	15
5	16	18	20	22	24	26	28	30
6	32	36	40	44	48	52	56	60
7	64	72	80	88	96	104	112	120

The three-bit EXP and FRAC fields are also used to form floating or double-floating operands (Figure 2-26). (Bits (63:32) are not present in single-precision floating point operands.)

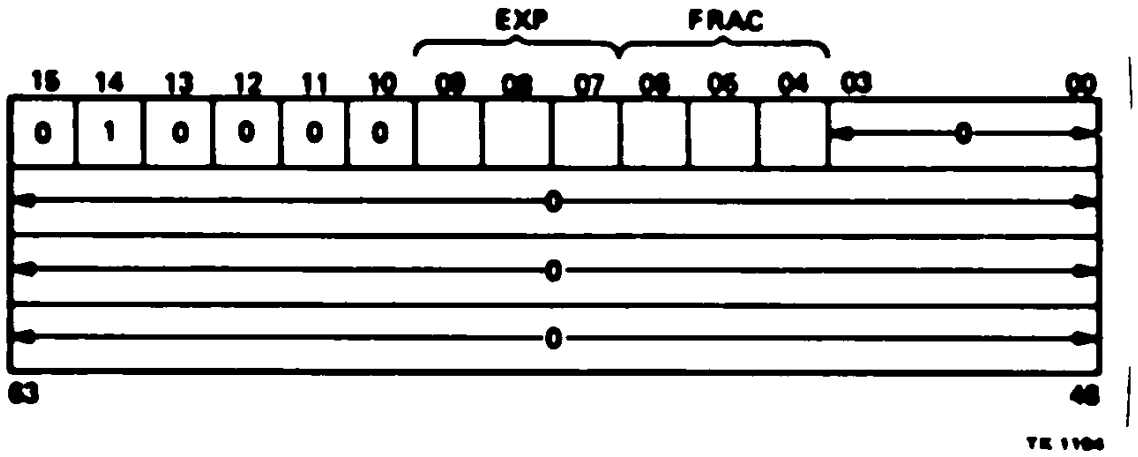


Figure 2-26 Literal Fields in Floating/Double-Floating Operands

2.3.3.6 Summary of Program Counter Addressing Modes – Table 2-17 summarizes the program counter addressing modes and includes the following information: the mode field value for each operand specifier, the assembler notation, the indexable modes, the operand access types for each mode. It also shows the results of using the stack pointer (R14) or program counter (R15) in each mode.

Table 2-17 Summary of Program Counter Addressing Modes

Mode Field Hex	Decimal	Name	Assembler	Access r	m	w	a	v	Indexable?
0–3	0–3	literal	S^#literal	y	f	f	f	f	f
8	8	immediate	I^#constant	y	u	u	y	y	y
9	9	absolute	@#address	y	y	y	y	y	y
A	10	byte relative	B^address	y	y	y	y	y	y
B	11	byte relative deferred	@B^address	y	y	y	y	y	y
C	12	word relative	W^address	y	y	y	y	y	y
D	13	word relative deferred	@W^address	y	y	y	y	y	y
E	14	longword relative	L^address	y	y	y	y	y	y
F	15	longword relative deferred	@L^address	y	y	y	y	y	y

Legend:
- f - reserved addressing mode fault
- u - Unpredictable
- y - yes, always valid addressing mode
- r - read access
- m - modify access
- w - write access
- a - address access
- v - variable bit field access

2.3.4 Index Addressing

In index mode addressing, the operand specifier shown in Figure 2-27 consists of two bytes: a primary operand specifier (bits ⟨07:00⟩) and secondary operand specifier (bits ⟨15:08⟩).

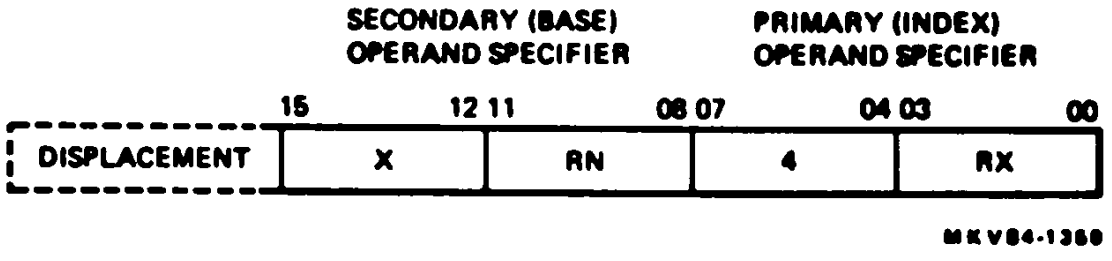


Figure 2-27 Index Mode Operand Specifier

The primary (index) operand specifier includes the index mode specifier of 4 and selects an index register [Rx] The secondary (base) operand specifier selects a base register (Rn) or the program counter (FF) and includes a specifier for one of the normal addressing modes (Index mode names are taken from the addressing mode of the secondary operand specifier) The base and index registers store the following information

- (Rn) contains the base address of a table of values
- [Rx] contains an index value to access sequential entries in the table
- When the PC is used, an absolute address or offset immediately follows the base operand specifier.

The following restrictions are placed on any general register used as an index register

- The PC cannot be used as an index register A reserved addressing mode fault occurs if attempted
- If the secondary operand specifier uses an addressing mode that results in register modification (autoincrement or autodecrement), the same register cannot be used as the index register. If it is, the primary operand address is unpredictable

The main advantage of index mode addressing is to provide efficient accessing of arrays. VAX-11 architecture provides for context indexing where the number in the index register is shifted left by the specified data type (once for byte, twice for word, three times for longword, four times for quadword, eight times for octaword) This allows loop control variables to be used in the address calculation without first shifting them the required number of times, thus reducing the number of instructions.

The following paragraphs briefly describe the allowable indexed modes Any attempt to use the register, index, or literal mode in the base operand specifier results in an illegal addressing mode fault.

2.3.4.1 Register Deferred Indexed Mode - The contents of [Rx] are first multiplied by the size (in bytes) of the operand (1 for byte, 2 for word, 4 for longword, 8 for quadword, 16 for octaword). This adjusted value is then added to the base address held by (Rn) The result is used as the operand address.

2.3.4.2 Autoincrement Indexed Mode - Operation is the same as for register deferred indexed However, the contents of (Rn) are incremented by the data type (1, 2, 4, 8, 16) after the operation.

2.3.4.3 Autodecrement Indexed Mode - The contents of (Rn) are first decremented by the data type (1, 2, 4, 8, 16) Operation is then the same as for register deferred indexed

2.3.4.4 Autoincrement Deferred Indexed Mode - Operation is the same as for autoincrement indexed. However, the result is used as the address of the base address of the data structure This mode is useful for accessing sequential lists of base addresses

2.3.4.5 Immediate Indexed Mode - This mode is recognized by the assembler, but is not generally useful because it indexes into the instruction stream (The PC contents point to the assumed base address.)

2.3.4.6 Absolute Indexed Mode - The absolute base address of the data structure immediately follows the secondary operand specifier in the instruction stream This base value is added to the adjusted value of [Rx] to obtain the address of the operand

2.3.4.7 Relative Indexed Mode - The offset that follows the secondary operand specifier is added to the updated PC (the address of the byte that follows the offset) The result is then added to the adjusted value of [Rx] to obtain the address of the operand

2.3.4.8 Relative Deferred Indexed Mode - The offset that follows the secondary operand specifier is added to the updated PC (the address of the byte that follows the offset) This result selects an address in a list of addresses The contents of that address are then added to the adjusted value of [Rx] to obtain the operand address

2.3.4.9 Displacement Indexed Mode - A displacement value (byte, word, or longword as specified by the data type) immediately follows the secondary operand specifier The displacement value is added to the contents of (Rn) and the contents of [Rx] are multiplied by the data size Both results are then added to produce the operand address

In this mode, (Rn) specifies the first base address in a list of base addresses The displacement value selects the base address for the desired data structure and [Rx] indexes into the list of operands.

2.3.4.10 Displacement Deferred Indexed Mode - A displacement value (byte, word, or longword, as specified by the data type) immediately follows the secondary operand specifier The displacement value is added to the contents of (Rn) and the result selects a base address in a list of base addresses The contents of [Rx] are multiplied by the data size The contents of the base address list location are then added to the adjusted value of [Rx] to produce the operand address.

2.3.4.11 Summary of Index Addressing Modes - Table 2-18 lists the valid index addressing modes in three groups and provides the assembler notations The first two groups use a general register or the PC as the base register The displacement modes specify general registers for the index and base registers and include the PC in their format to provide the displacement values

2.4 NATIVE MODE INSTRUCTIONS

The VAX native mode instruction set consists of over 200 opcodes that are listed in the *VAX Architecture Handbook* (EB-19580) The *Handbook* also provides further details and descriptions of how the instructions affect the PSL condition codes.

This section summarizes the native mode instructions according to their class or function Appendixes A and B list all instructions and their mnemonics in opcode order

Many instructions have two-operand and three-operand forms that eliminate moving the data to or from temporary locations The two-operand forms store the result in one of the two operands. The three-operand forms use two values to calculate a third which is then stored in a third operand location. Table 2-19 contains the terms used in discussions that refer to data sizes.

Table 2-18 Summary of Index Addressing Modes

Addressing Mode	Assembler Notation	
General Register Modes		
Register Deferred Indexed	(Rn)[Rx]	
Autoincrement Indexed	(Rn)+[Rx]	
Autodecrement Indexed	-(Rn)[Rx]	
Autoincrement Deferred Indexed	@(Rn)+[Rx]	
Program Counter Modes		
Immediate Indexed	#n[Rx]	(Recognized by assembler but not generally useful because of indexing into instruction stream.)
Absolute Indexed	@#A[Rx]	
Relative Indexed	A[Rx]	(General notation)
	B^A[Rx] W^A[Rx] L^A[Rx]	
Relative Deferred Indexed	@A[Rx]	(General notation)
	@B^A[Rx] @W^A[Rx] @L^A[Rx]	
*Displacement Modes**		
Displacement Indexed	D(Rn)[Rx]	(General notation)
	B^D(Rn)[Rx] W^D(Rn)[Rx] L^D(Rn)[Rx]	
Displacement Deferred Indexed	@D(Rn)[Rx]	(General notation)
	@B^D(Rn)[Rx] @W^D(Rn)[Rx] @L^D(Rn)[Rx]	

* Any attempt to use register, index, or literal mode in the base operand specifier results in an illegal addressing mode fault.

Table 2-19 Summary of Data Terms

Term and Meaning		Data Size		Sign Bit	
B	= Byte	8	bits	Bit	⟨7⟩
W	= Word	16	bits	Bit	⟨15⟩
L	= Longword	32	bits	Bit	⟨31⟩
Q	= Quadword	64	bits	Bit	⟨63⟩
O	= Octaword	128	bits	Bit	⟨127⟩
F	= F_floating	32	bits	Bit	⟨15⟩
D	= D_floating	64	bits	Bit	⟨15⟩
G	= G_floating	64	bits	Bit	⟨15⟩
H	= H_floating	128	bits	Bit	⟨15⟩
P	= Packed decimal	—		—	

2.4.1 Logical, Stack, and Address Instructions

Table 2-20 lists the instructions that apply to the integer and floating-point data types.

2.4.1.1 Logical Integer Instructions

The logical integer instructions operate as follows.

Rotate Longword — ROTL rotates the source operand by the number of bits specified by the count operand and places the result in the destination operand. A positive count operand rotates to the left, while a negative count operand rotates to the right. A count operand of 0 replaces the destination with the source operand.

Move — MOV moves the source operand to the destination operand address. The source and destination are the same length. The source may be specified using the short literal or immediate mode to place a constant in the destination.

Move Complemented/Move Negated — MCOM stores the one's complement form of the source in the destination, while MNEG stores the two's complement form.

Move Zero-Extended — MOVZ extends the size of the operand by supplying zeros in the extra, high-order bytes.

Convert — CVT converts the data size of the source operand to the data size of the destination. For conversion from a shorter to a longer data type, the value is sign-extended For conversion from a longer to a shorter data type, the higher order bytes are truncated.

Clear — CLR zeros the destination operand.

Compare — CMP subtracts the second operand from the first and reports the results in the condition codes of the PSL.

Test — TST compares an operand for a less than zero (negative) or equal to zero value.

Table 2-20 Logical, Stack, and Address Instructions

Mnemonic	Instruction Name
Logical Integer Instructions	
ROTL	Rotate longword
MOV__	Move (B, W, L, Q, O, F, D, G, H)
MNEG__	Move negated (B, W, L, F, D, G, H)
MCOM__	Move complemented (B, W, L)
MOVZ__	Move zero-extended (BW, BL, WL)
CVTB__	Convert byte to (W, L, F, D, G, H)
CVTW__	Convert word to (B, L, F, D, G, H)
CVTL__	Convert longword to (B, W, F, D, G, H)
CVTF__	Convert F__floating to (B, W, L, D, G, H)
CVTD__	Convert D__floating to (B, W, L, F, H)
CVTG__	Convert G__floating to (B, W, L, F, H)
CVTH__	Convert H__floating to (B, W, L, F, D, G)
CVTR__L	Convert rounded (F, D, G, H__floating) to longword
CLR__	Clear (B, W, L, Q, O, F, D, G, H)
CMP__	Compare (B, W, L, F, D, G, H)
TST__	Test (B, W, L, F, D, G, H)
Logical Bit Instructions	
BIS__2	Bit set (B, W, L) 2-operand
BIS__3	Bit set (B, W, L) 3-operand
BIC__2	Bit clear (B, W, L) 2-operand
BIC__3	Bit clear (B, W, L) 3-operand
BIT__	Bit test (B, W, L)
XOR__2	Exclusive OR (B, W, L) 2-operand
XOR__3	Exclusive OR (B, W, L) 3-operand
Stack and Address Instructions	
MOVA__	Move address of (B, W, L, Q, O, F, D, G, H)
PUSHA__	Push address of (B, W, L, Q, O, F, D, G, H)
PUSHL	Push longword
PUSHR	Push register
POPR	Pop register

2.4.1.2 Logical Bit Instructions - The logical bit instructions are as follows

Bit Set — BIS, when a two-operand instruction, inclusive-ORs the mask operand with the destination and stores the results in the destination The three-operand form inclusive-ORs the mask operand with the source operand and stores the results in the destination

Bit Clear — BIC, when a two-operand instruction, ANDs a one's complement of the mask operand with the destination operand and stores the results in the destination The two-operand form ANDs a one's complement of the mask operand with the source operand and stores the results in the destination.

Exclusive OR XOR, when a two-operand instruction, exclusive-ORs the mask operand with the destination operand, storing the results in the destination. The three-operand form exclusive-ORs the mask operand with the source operand, storing the results in the destination.

Bit Test — BIT ANDs the mask operand with the source operand, and reports the results in the condition codes of the PSL Neither operand is affected

2.4.1.3 Stack and Address Instructions - These instructions fetch and store addresses without accessing the data, allowing the user program to save or load general registers in one operation

Move Address MOVA stores the address of a datum of specified size in a specified register or memory location

Push Address PUSHA stores the address on the stack

Push Longword — PUSHL pushes a selected longword on the stack This instruction is the same as a move longword (MOVL), but is one byte shorter, using the stack pointer in register deferred mode.

Push Registers — PUSHR pushes a selected set of general registers on the stack A mask word must be supplied in which each set bit (14:00) represents a register (R14:R00) to be saved on the stack. The only register that cannot be saved using this instruction is the PC (R15)

Pop Registers — POPR reverses the PUSHR operation, loading each register from successive longwords in the stack according to a mask word The PUSHR and POPR instructions may be used in place of a sequence of Move instructions that save and restore registers when entering or exiting a subroutine.

2.4.2 Integer Arithmetic and Floating-Point Instructions
Table 2-21 lists the instructions that perform arithmetic operations on integer or floating-point data

2.4.2.1 Arithmetic Instructions - The two-operand arithmetic instructions perform the specified operation and store the result in one of the two operands, for example: "Add the (B, W, or L) value in location A to the value in B and store the result in B." The three-operand instructions use two values to calculate a third, for example "Add the (B, W, or L) value in location A to the value in B and store the result in C." The three-operand instructions apply to integer and floating point data but equivalent instructions are also provided for packed decimal data. In the floating formats, the product operand result is rounded for both two- and three-operand forms.

2.4.2.2 Extended-Precision Instructions - Arithmetic instructions are also provided that extend the accuracy for repeated computations.

Table 2-21 Integer Arithmetic and Floating-Point Instructions

Mnemonic	Instruction Name
INC_	Increment (B, W, L)
DEC_	Decrement (B, W, L)
ASH_	Arithmetic shift (L, Q)
ADD_2	Add (B, W, L, F, D, G, H) 2-operand
ADD_3	Add (B, W, L, F, D, G, H) 3-operand
ADWC	Add with carry
ADAWI	Add aligned word, interlocked
SUB_2	Subtract (B, W, L, F, D, G, H) 2-operand
SUB_3	Subtract (B, W, L, F, D, G, H) 3-operand
SBWC	Subtract with carry
MUL_2	Multiply (B, W, L, F, D, G, H) 2-operand
MUL_3	Multiply (B, W, L, F, D, G, H) 3-operand
DIV_2	Divide (B, W, L, F, D, G, H) 2-operand
DIV_3	Divide (B, W, L, F, D, G, H) 3-operand
EMUL	Extended multiply
EDIV	Extended divide
EMOD_	Extended modulus (F, D, G, H)
POLY_	Polynomial evaluation (F, D, G, H)

Extended Multiply — EMUL uses longword integer arguments and produces a quadword result EMUL also implements high-level language statements, such as: "Set D equal to (A times B) plus C."

Extended Divide — EDIV divides a quadword integer by a longword and produces a longword quotient with a longword remainder.

Extended Modulus — EMOD multiplies F_ or D_floating point numbers with an extended-precision floating point number which it extends by 8 bits for an effective 9 or 19 digits of accuracy It then returns an integer portion with a separate fractional portion This instruction is useful for preserving the precision of the inputs throughout the trigonometic and exponential evaluations.

Polynomial Evaluation — POLY evaluates a polynomial from a table of coefficients using Horner's method This instruction is used extensively in the high-level languages math library for operations such as sine and cosine functions

2.4.3 Character String Instructions

Table 2-22 lists the character string instructions that operate on strings of contiguous bytes of characters or data

Table 2-22 Character String Instructions

Mnemonic	Instruction Name
MOVC_	Move character (three- or five-operand)
MOVTC	Move translated characters
MOVTUC	Move translated until character
CMPC_	Compare characters (three- or five-operand)
LOCC	Locate character
SKPC	Skip character
MATCHC	Match characters
SCANC	Scan for character
SPANC	Span characters
CRC	Cyclic redundancy check

2.4.3.1 Move and Compare Instructions

Move Character — MOVC copies a character string from one block location to another for optimum block transfer operations The three-operand form moves a character string between two locations of equal length The five-operand form moves a string between locations of unequal length, supplying specified fill characters if the destination address is larger than the source address

Move Translated Characters/Move Translated Until Character — MOVTC and MOVTUC use six operands to translate the string from one form to another while moving it Both instructions use the source string as a list of offsets into a translation table, reading characters from the table selected by the offset list, and storing them in the destination address

Both instructions move strings of unequal lengths The MOVTC instruction allows a fill character to be supplied which it uses (untranslated) to pad the destination string locations to the given size The MOVTUC instruction allows any number of escape characters be stored in the table When the translated offset reads an escape character from the table, translation stops

Compare Character — CMPC performs a character-by-character comparison of byte strings The three-operand and five-operand forms each compare two strings from beginning to end Each acknowledges when it reaches the first character difference between the strings or when it reaches the end of either string The five-operand form, however, allows a fill character to be supplied, which it uses when comparing strings of unequal length

2.4.3.2 Locate, Skip, and Match Instructions

Locate Character/Skip Character — LOCC and SKPC search for single characters in a string. LOCC searches a given string for a character that matches the supplied search character. LOCC is useful, for example, when searching for the delimiter at the end of a variable length string. SKPC, however, finds the first character in the string that is different from the search character supplied. SKPC is useful for skipping through fill characters at the end of a field to find the beginning of the next field.

Match Characters — MATCHC is similar to the locate character instruction but locates multiple character substrings. MATCHC searches a string for the first occurrence of a supplied substring.

2.4.3.3 Scan, Span, and CRC Instructions

Span Characters/Scan for Characters — SPANC and SCANC search for character types or classes. For these instructions, the following data must be supplied: a character string, a mask, and the base address of a 256-byte table of character type definitions. For each character in a given string, the instruction looks up the type code in the table for that character. It then ANDs the given mask with the character's type code. SPANC finds the first character in the string of the type indicated by the mask. SCANC finds the first character in the string of any other type than the one indicated by the mask.

Cyclic Redundancy Check — CRC calculates a cyclic redundancy check for a given data string using any CRC polynomial up to 32 bits long. The operating system library includes tables for standard CRC functions (such as CRC-16).

2.4.4 Packed Decimal String Instructions

Many of the operations for integer and floating point data discussed in Section 2.4.1 also apply to packed decimal strings. The packed decimal class consists of the instructions listed in Table 2-23.

Table 2-23 Packed Decimal String Instructions

Mnemonic	Instruction Name
CVTPL	Convert packed to longword
CVTPS	Convert packed to leading separate numeric
CVTPT	Convert packed to trailing numeric
CVTLP	Convert longword to packed
CVTSP	Convert leading separate numeric to packed
CVTTP	Convert trailing numeric to packed
MOVP	Move packed
CMPP_	Compare packed (three- or four-operand)
ASHP	Arithmetic shift and round packed
ADDP_	Add packed (four- or six-operand)
SUBP_	Subtract packed (four- or six-operand)
MULP	Multiply packed
DIVP	Divide packed
EDITPC	Edit packed to character string

Convert Packed — CVTP__ and CVT__P perform conversions between packed decimal and the commonly used numeric formats

Move Packed — MOVP copies a packed decimal string from one location to another

Compare Packed — CMPP compares two packed decimal strings in two variations a three-operand form (CMPP3) for strings of equal lengths and a four-operand form (CMPP4) for strings of different lengths

Arithmetic Shift Packed — ASHP scales a packed decimal string up or down by a given power of 10 while moving it and optionally rounding the value

Add Packed/Subtract Packed — ADDP and SUBP add and subtract two packed decimal strings with the option of replacing the addend or subtrahend with the result (ADDP4 and SUBP4), or storing the result in a third string (ADDP6 or SUBP6)

Multiply Packed/Divide Packed — MULP and DIVP multiply and divide two packed decimal strings and store the result in a third string The numeric with trailing sign format allows the use of various sign and data encodings including the zoned and overpunched formats described in Section 2 1 4 3

Edit Packed — EDITPC is a special packed-decimal-to-character-string conversion instruction that provides formatted numeric output functions The operands are the source length, source address, pattern address, and destination address

The instruction edits the input string according to operators stored in the pattern string and continues until the END operator is found The pattern string consists of one-byte operators some that take no operand, some that take an operand following the operator, and some that take a repeat count in the low-order nibble

The instruction performs the conversion using the pattern operators to create a numeric output string with the following characteristics

- Leading zero fill
- Leading zero protection
- Leading character fill protection
- Floating sign insertion
- Floating currency symbol
- Special sign representation
- Character insertion
- Blank field when zero

After execution, general registers R0 through R5 contain the resulting length and address information

2.4.5 Variable-Length Bit Field Instructions

These instructions allow the definition, access, and modification of fields whose size and location can be specified. The variable-length bit field instructions are listed in Table 2-24.

Table 2-24 Variable-Length Bit Field Instructions

Mnemonic	Instruction Name
INSV	Insert field
EXTV	Extract field
EXTZV	Extract zero-extended field
CMPV	Compare field
CMPZV	Compare zero-extended field
FFS	Find first set bit
FFC	Find first clear bit

The bit field location is determined from a base address or from a register with a signed bit offset. If the bit field is in memory, the offset range can be as large as 2^{32} minus 1 (approximately 16 million bytes). If the field is in a register, the offset can be as large as 31. Bit fields of arbitrary lengths (0 to 32 bits) may be used to store status codes or to compactly store data structure header information.

Insert Field/Extract Field — INSV and EXTV store and retrieve bit field data. INSV stores data in a field by taking a specified number of bits of a longword (starting from the low-order bit) and writing them into a field that may start at any bit, relative to a given base address. The field can then be retrieved as either a signed (EXTV) or unsigned (EXTZV) field.

Compare Field/Find First — CMPV, FFS, and FFC test the contents of a bit field. CMPV first extracts a field then compares it with a given longword. The field may be interpreted as signed (CMPV) or unsigned (CMPZV). The find first instructions locate the first bit in a field that is clear (FFC) or is set (FFS), scanning from the low-order to the high-order bit.

These instructions are useful for scanning a status control longword. For example, the longword may represent a set of queues processed in order by priority 0 (high) to 31 (low) where each set bit represents an active queue. The find first set (FFS) instruction quickly returns the highest priority queue that is active. The find first instructions are also useful for scanning an allocation table (bit map) of arbitrary length when used with the SKPC instructions.

2.4.6 Queue Instructions

Queues consist of circular, doubly linked lists of data items that may be manipulated using the queue instructions. The two instructions that allow the construction and maintenance of queue data structures are listed in Table 2-25.

Table 2-25 Queue Instructions

Mnemonic	Instruction Name
INSQUE	Insert into queue
INSQHI	Insert into queue head, interlocked
INSQTI	Insert into queue tail, interlocked
REMQUE	Remove from queue
REMQHI	Remove from queue head, interlocked
REMQTI	Remove from queue tail, interlocked

The first longword of a queue entry contains the forward pointer to the next entry in the queue. The next longword contains the backward pointer to the preceding entry. One queue entry is arbitrarily treated as the head of the queue. Since a list is circular, the tail of a queue is the entry that points to the head of the queue. In practice, the first entry of a queue is a permanently allocated listhead that contains only the pointers to the first and last elements.

Insert into Queue — INSQUE inserts entries into queues. If an entry is the first item in a queue, INSQUE creates the queue.

Remove from Queue — REMQUE removes entries from a queue and deletes the queue if the entry is the last item removed. Entries may be inserted or removed at the head or tail of a queue or anywhere in between.

Cooperating processes can access a queue at the same time without external synchronization. However, if more than one process is allowed to access a given queue at the same time, each process should insert or remove entries only at the head or tail of the queue. If only one process at a time accesses a queue, entries can be inserted or removed anywhere in the queue.

2.4.7 Branch and Jump Instructions

Table 2-26 lists the instructions that transfer program control to another memory location by loading a new address to the program counter.

2.4.7.1 Unconditional Branch and Jump Instructions

Branch with Displacement - BRB or BRW, for an unconditional branch a byte or word displacement (offset) value is sign-extended and added to the current value of the program counter to obtain the new address. BRB allows branches of -128 to +127 bytes, and BRW allows branches of -32,768 to +32,767 bytes, from the current location. BRB and BRW are mainly used to relocate the program to addresses relatively close to the current instruction.

Jump - JMP loads a new address to the program counter using one of the normal addressing modes and is generally used for relocations to addresses farther away from the current location. The destination is stored as an address operand and any addressing mode may be used except the register, immediate, or short literal mode.

Table 2-26 Branch and Jump Instructions

Mnemonic	Instruction Name
Unconditional Branch and Jump Instructions	
BR__	Branch with (B, W) displacement
JMP	Jump
Branch-on-Bit Instructions	
BB__	Branch on bit (set, clear)
BBS__	Branch on bit set and (set, clear) bit
BBC__	Branch on bit clear and (set, clear) bit
BLB__	Branch on low bit (set, clear)
BBSSI	Branch on bit set and set, interlocked
BBCCI	Branch on bit clear and clear, interlocked
Branch-on-Condition Instructions	
BLSS	Branch on less
BLEQ	Branch on less or equal
BEQL	Branch on equal
BNEQ	Branch on not equal
BGTR	Branch on greater
BGEQ	Branch on greater or equal
BLSSU	Branch on less, unsigned
BLEQU	Branch on less or equal, unsigned
BEQLU	Branch on equal, unsigned
BNEQU	Branch on not equal, unsigned
BGTRU	Branch on greater, unsigned
BGEQU	Branch on greater or equal, unsigned
BCS	Branch on carry set
BCC	Branch on carry clear
BVS	Branch on overflow set
BVC	Branch on overflow clear
Loop Control and Case Instructions	
AOBLSS	Add one and branch if less
AOBLEQ	Add one and branch if less or equal
SOBGTR	Subtract one and branch if greater
SOBGEQ	Subtract one and branch if greater or equal
ACB__	Add compare and branch (B, W, L, F, D, G, H)
CASE__	Case on multiple conditions (B, W, L)

2.4.7.2 Branch-on-Bit Instructions – The branch-on-bit instructions test any specified bit of a longword, then branch with byte displacement if the following conditions exist:

Branch on Bit Set or Clear – BBS or BBC branch if the specified bit is set or clear.

Branch on Bit Set or Clear and Set or Clear the Bit – BBSS, BBSC, BBCS or BBCC set or clear the specified bit, then branch if it was set or clear before the operation. This is useful for keeping track of procedure completion or initialization, and for signaling the completion or initialization of a procedure to a cooperating process

Branch on Low Bit Set or Clear BLBS or BLBC branch if the low bit (bit ⟨00⟩) is set or clear is useful for testing for odd or even integer values (signed or unsigned), or for false (zero) or true (one) FORTRAN compilations

Branch-on-Bit Interlocked BBSSI and BBCCI operate the same as BBSS and BBCC but provide control variable protection. The SBI provides a memory interlock on these instructions to prevent an interlocked instruction from one device altering a bit while another device is testing it.

2.4.7.3 Branch-on-Condition Instructions – The branch-on-condition instructions transfer program control to another location depending on the states of the condition codes in the processor status word (PSW). Three instruction groups branch on the following conditions.

- *Signed* – Test combinations of the N and Z condition codes in the PSW; used to test the results of instructions that operate on integer and field data types, treating them as signed integers, floating point data, or decimal strings. Signed conditional branches occur on the tests shown in Table 2-27.

- *Unsigned* – Test combinations of the C and Z condition codes in the PSW; used to test the results of instructions that operate on integer and field data types, treating them as unsigned integers, character strings, or addresses. Unsigned conditional branches occur on the tests shown in Table 2-28.

- *Carry and Overflow* – Test the C and V condition codes in the PSW; used for checking overflows when traps are not enabled, or checking for a carry, borrow, or overflow from an integer, floating, or packed decimal operation. Carry and overflow conditional branches occur on the tests shown in Table 2-29

Table 2-27 Branch-on-Condition Instructions (Signed)

Mnemonic	Name	Branch-on-Condition Code
BLSS	Branch on less	If N equals 1
BLEQ	Branch on less or equal	If N or Z equal 1
BEQL	Branch on equal	If Z equals 1
BNEQ	Branch on not equal	If Z equals 0
BGTR	Branch on greater	If N or Z equal 0
BGEQ	Branch on greater or equal	If N equals 0

Table 2-28 Branch-on-Condition Instructions (Unsigned)

Mnemonic	Name	Branch-on-Condition Code
BLSSU	Branch on less, unsigned	If C equals 1
BLEQU	Branch on less or equal, unsigned	If C or Z equal 1
BEQLU	Branch on equal, unsigned	If Z equals 1
BNEQU	Branch on not equal, unsigned	If Z equals 0
BGTRU	Branch on greater, unsigned	If C or Z equal 0
BGEQU	Branch on greater or equal, unsigned	If C equals 0

Table 2-29 Branch-on-Condition Instructions (Carry or Overflow)

Mnemonic	Name	Branch-on-Condition Code
BCS	Branch on carry set	If C equals 1
BCC	Branch on carry clear	If C equals 0
BVS	Branch on overflow set	If V equals 1
BVC	Branch on overflow clear	If V equals 0

2.4.7.4 Loop Control and Case Instructions - The following three types of branch instructions can be used to write efficient loops and require byte displacements to be supplied:

- Loop instructions that increment a counter, compare it with a limit value, and branch until the result equals or exceeds the set limit
- Loop instructions that decrement a counter and branch until the counter is zero or negative
- Computed branch instructions that branch on computed values

Add One and Branch - The first type provides a basic add-one-and-branch loop. A set value must be supplied along with a counter variable that is incremented each time the loop is executed. In the add-one-and-branch-if-less-than (AOBLSS) instruction, the loop repeats until the counter equals the set limit. In the add-one-and-branch-if-less-than-or-equal (AOBLEQ) instruction, the loop repeats until the counter exceeds the set limit.

Subtract One and Branch - The second type provides a basic subtract-one-and-branch loop. A counter variable must be supplied that is incremented or decremented each time the loop is executed. In the subtract-one-and-branch-if-greater-than (SOBGTR) instruction, the loop repeats until the counter equals zero. In the subtract-one-and-branch-if-greater-than-or-equal (SOBGEQ) instruction, the loop repeats until the counter goes negative.

Add Compare and Branch/Case on Conditions - The third type includes the add-compare-and-branch (ACB) instruction (with B, W, L, F, D, G, or H offset), and the case-on-multiple conditions (CASE) instruction (with B, W, or L offset).

The ACB instruction efficiently implements the FORTRAN language DO statement and the BASIC language FOR statement. Counter and step (addend) values must be supplied. The instruction adds the step value to the counter on each execution of the loop. The resulting sign of the step value then

determines the logical comparison. loop on (branch until) a less-than-or-equal comparison (the step value is positive); or loop on (branch until) a greater-than-or-equal comparison (the step value is negative).

The CASE instruction implements higher-level language computed GO TO statements and allows one of many similar sections to be selected by a single value called the selector. The base address (lowest legal value) to a list of word-length displacements must be supplied along with the upper limit (highest legal value).

The selector input value (B, W, or L) is adjusted by subtracting the base value. The difference indexes into the table of word length displacements. If the result is less than zero or is greater than the upper limit, no branch occurs. If the adjusted select value is legal (falls within the limits of the table) the selected word length displacement is added to the PC and the branch is taken.

2.4.8 Branch/Return and Jump Subroutine Instructions

Table 2-30 lists instructions that are provided for entering and leaving subroutines.

Table 2-30 Branch/Return and Jump Subroutine Instructions

Mnemonic	Instruction Name
BSB_	Branch to subroutine (B, W)
RSB	Return from subroutine
JSB	Jump to subroutine

Branch to Subroutine - BSB must supply a byte or word displacement. It saves the contents of the program counter on the stack before loading the program counter with the new address.

Jump to Subroutine - JSB uses one of the normal addressing modes and does not save the program counter.

Return from Subroutine - RSB pops the first longword off the stack and loads it back to the program counter. Very efficient programs may be written using subroutines using the BSB instruction, which may be two or three bytes long, and the RSB instruction which is only one byte.

2.4.9 Procedure Call and Return Instructions

Procedures are general purpose routines that pass argument lists between the user and the operating system. They are an extremely efficient way to ensure that registers are saved across procedure calls. The procedure instructions are listed in Table 2-31.

Table 2-31 Procedure Call and Return Instructions

Mnemonic	Instruction Name
CALLG	Call procedure with general argument list
CALLS	Call procedure with stack argument list
RET	Return from procedure

The CALL instructions provide language processors and the operating system with a standard procedure calling interface as follows

- Store only the registers used by a procedure before entering the procedure.
- Pass an argument list to a procedure
- Maintain stack, frame, and argument pointer registers.
- Initialize arithmetic trap enables to a given state

Call Procedure with General Argument List CALLG accepts and passes the argument list address to the procedure in the argument pointer register

Call Procedure with Stack Argument List — CALLS passes any argument list that was placed on the stack by loading the stack address to the argument pointer register.

When a call procedure instruction is issued, it must supply the address of the procedure. The first word of a procedure contains an entry mask that is used the same as the entry mask defined for the push register (PUSHR) instruction Each of the 12 low-order bits in the word represents one of the general registers R0 through R11 that the procedure is to use. R0 and R1, however, are reserved for use by the procedure to return status or function values to the calling routine.

The CALL instructions examine the first word and save the indicated registers on the stack (call frame or stack frame) They also store the old contents of the argument pointer (AP), frame pointer (FP), and program counter (PC) registers Each procedure may subsequently call another procedure. This is called "nesting of procedures" and each call frame is then located using the frame pointer.

Return from Procedure Instruction — RET is issued by a procedure when it completes its execution RET uses the frame pointer register to find the saved registers to be restored and to clean up any data left on the stack including any nested routine linkages. A procedure may return values by using the argument list or other registers.

2.4.10 Special Purpose Instructions

The native mode includes special purpose instructions, which are listed in Table 2-32.

Table 2-32 Special Purpose Instructions

Mnemonic	Instruction Name
NOP	No operation
MOVPSL	Move processor status longword
BISPSW	Bit set processor status word
BICPSW	Bit clear processor status word
INDEX	Compute index

No Operation — NOP is useful for reserving memory areas or for debugging or making changes to the software

Move Processor Status Longword — MOVPSL allows examination of the PSL contents by loading them to a specified location

Bit Set/Bit Clear Processor Status Word — BISPSW and BICPSW allow the setting or clearing of the condition code and trap enable bits in the PSW field of the PSL

Compute Index — INDEX calculates an index for an array of fixed length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments a subscript; lower and upper subscript bounds, an array element size; a given index; and a destination for the calculated index The calculation incorporates range checking for high-level languages using subscript bounds It optimizes index calculation by removing invarient expressions.

2.4.11 Privileged Instructions

The privileged instructions are only executable in the kernel mode at the operating system level When issued by a process in other than kernel mode, each may generate a reserved opcode fault to the operating system which then determines the access privilege of the requesting process Privileged instructions are listed in Table 2-33

Table 2-33 Privileged Instructions

Mnemonic	Instruction Name
CHM__	Change mode to (K, E, S, U)
PROBER	Probe read access
PROBEW	Probe write access
REI	Return from exception or interrupt
SVPCTX	Save process context
LDPCTX	Load process context
MTPR	Move to privileged register
MFPR	Move from privileged register
XFC	Extended function call
BPT	Breakpoint fault
BUG__	Bugcheck (W, L)
HALT	Halt processor

User mode software can obtain privileged operating system services, without compromising the integrity of the system, by using a standard CALL instruction.

Change Mode — CHM is issued by the operating service dispatcher before entering the procedure CHM allows an access mode change to take place only to the same or a more privileged mode When the mode change takes place, the previous mode is saved in the previous mode field of the PSL, allowing the more privileged program to determine the privilege of its caller.

CHM is a special trap instruction that is used as an operating system service call instruction. User mode software can issue CHM instructions but, because the operating system receives the trap, nonprivileged users cannot write code to execute in any of the privileged access modes. However, user mode software may include a condition handler that changes mode to user traps. CHM is useful for providing general services for user mode software. (Actually, the system manager grants all privileges to write any code that handles change mode traps to more privileged access modes.)

Probe Read Access and Probe Write Access — For service procedures written to execute in the privileged access modes (kernel, executive, and supervisor), the processor provides address access privilege validation instructions.

PROBER and PROBEW allow a procedure to quickly check the read and write access protection of pages in memory against the privileges of the caller requesting access to a particular location. This allows the operating system to provide services that execute in privileged modes to less privileged callers and still prevent the caller from accessing protected areas of memory.

Return from Exception or Interrupt — REI is used to exit the operating system privileged service procedures and the interrupt and exception service routines. REI is the only way that the privilege of the processor access mode can be lowered. Like the procedure and subroutine return instructions, REI restores the program counter and the processor state to resume the process at the point where it was interrupted.

REI performs special services, however, that the normal return instructions do not. Following the interrupt or exception service routine, for example, REI checks for any asynchronous system traps that may have queued for the current process (while the service routine was executing) to ensure that the process receives them.

REI also checks to ensure that the mode it is returning control to is of the same or less privilege than the mode that the processor was in when the exception or interrupt occurred. REI is available to all software, including user-written trap handling routines. However, a program that attempts to increase its privilege by altering the processor state to be restored, generates a reserved opcode fault.

Save Process Context — SVPCTX is used by the context switching procedure to save the current process context when the operating system schedules a context switch.

Load Process Context — LDPCTX is used to load another process context. The context switching procedure identifies the location of the next hardware context to be loaded by updating an internal processor register.

Move to/Move from Processor Register — MTPR and MFPR are the only instructions that can access the internal processor registers. These are privileged instructions that may be issued only in the kernel mode. (The privileged processor registers are listed in Section 2.1.5 3.)

Extended Function Call — XFC allows microcode escapes to customer-defined instructions in the writable control store.

Breakpoint Fault — BPT causes the processor to execute the kernel mode condition handler associated with the breakpoint fault exception vector. BPT is used by the operating system debugging utilities, but can also be used by any process that sets up a breakpoint fault condition handler.

Bugcheck — BUGW or BUGL is used under system diagnostic procedures to request a report of the accumulated software, hardware, and firmware errors from the error log facility maintained by the operating system It must supply a longword message identifier that identifies the kernel process (a word identifier is zero-extended to longword) When issued, it initiates an exception and asserts a vector to the processor For fatal errors, the operating software may elect to crash the system

Halt — HALT stops the processor It is issued by the operating system only in response to a fatal error or when performing a system shutdown by operator request

2.5 COMPATIBILITY MODE

The compatibility mode instruction set is much the same as for the PDP-11 It is provided as a convenient tool for users who are changing from a PDP-11 to a VAX system or who use both types of systems For most efficient operation, however, PDP-11 programs should be written in the native mode, whenever possible, or converted by recompiling on the VAX system Refer to Chapter 17 of the *VAX Architecture Handbook* (EB-19580) for a description of the compatibility mode instructions and their effect on VAX hardware registers

2.5.1 PDP-11 Instruction Execution

Under the control of the operating system, the processor executes PDP-11 instruction streams in the context of the current process The instructions execute exactly as on a PDP-11/70 processor running in user mode with the following exceptions:

- The floating point and extended instructions are illegal, as are the privileged instructions (HALT, WAIT, RESET, SPL, MARK, CSM, MFPT, MTPS, MFPS)

- The trap instructions (BPT, IOT, EMT, TRAP) cause the processor to leave compatibility mode and enter the native mode where the trap is serviced or the instruction is simulated

- The move previous instructions (MFPI, MTPI, MFPD, MTPD) do not trap to the native mode under VAX memory management and are illegal They do not exist in VAX and cannot be simulated by the operating system

  However, they execute as they would on a PDP-11 in user mode with instruction and data space overmapped Ignoring the previous access level, they act as PUSH and POP instructions and make direct reference to the stack

2.5.2 Operating System Restrictions

The operating system provides an environment for executing most user mode programs written for a PDP-11, except for stand-alone software However, a PDP-11 program cannot leave the user access mode and there are restrictions on the environment that the native operating system can provide

PDP-11 operating systems cannot run in compatibility mode All compatibility mode software is expected to rely on the services of the native operating system for I/O operations, interrupt and exception handling, and memory management (including memory allocation) all of which involve the use of privileged instructions VMS must therefore perform services normally handled by a PDP-11 operating system, such as RSX-11M or RSTS/E For PDP-11 images written to execute under RSX-11M, VAX/VMS provides an interface program called the application migration executive (AME) that acts as an interface, passing control to VMS when the program requests an RSX-11M executive directive When the service has been performed in native mode, VMS passes control back to the AME, which then passes it back to the PDP-11 program

A subset of the PDP-11 processor status word (PSW) is defined for the compatibility mode Only the condition codes and the trace trap bit are relevant to the PDP-11 instruction stream.

Interrupts and exceptions that occur in compatibility mode cause the processor to enter the native mode for the operating system to handle There are some exceptions that apply only to compatibility mode These include illegal instruction exceptions and odd address traps

2.5.3 PDP-11 Addressing Modes

PDP-11 addressing modes are available in the compatibility mode but PDP-11 addresses are 16-bit byte addresses There is a one-to-one relation between compatibility mode virtual addresses and the first 64K bytes of virtual address space available to native mode processes A compatibility mode program is restricted to reference only these addresses, although the operating system provides most of the PDP-11 memory management mechanisms For example, compatibility mode supports PDP-11 memory segment protection in 512-byte pages rather than in 64-byte segments

The general registers are available under the following criteria

- Compatibility mode registers R0 through R6 are bits (15 00) of the native mode general registers R0 through R6 Bits (31 16) are ignored
- Compatibility mode R7 (program counter) is bits (15 00) of native mode general register R15 (program counter)
- For an interrupt or exception in compatibility mode, the contents of native mode general register R7 are unpredictable, and bits (31.16) of native registers R0 through R6 and R15 are zero
- Native mode registers R8 through R14 are not affected by the compatibility mode General register R6 acts as the stack pointer (SP) for program-local temporary data storage The program-local stack has address space allocated in the program region, not in the control region However, R6 is not used by the hardware for exceptions or interrupts nor does it provide stack overflow trap protection
- Double-error traps do not exist in compatibility mode because the first error changes the processor to native mode Other trap conditions such as power failure, memory parity, and memory management change the processor to native mode and are not implemented in compatibility mode

CHAPTER 3

CHAPTER 3
LOGIC DESCRIPTION

3.1 GENERAL

This chapter provides a description of the following CPU logic to the block diagram level. It includes functional block diagrams that show the data and addressing paths and are intended for use with the maintenance print set.

- CPU Microprocessor
- Instruction Buffer and Decoder
- CPU Data Paths
- Exceptions and Interrupts
- System Clocks

Because the VAX-11 uses data formats of different byte lengths, data and address references are expressed in hexadecimal (hex), except where binary is used to clarify individual bit values.

3.1.1 CPU Busses

Figure 1-2 in Section 1.3 introduces the CPU logic. It also shows the interconnection of the following busses that handle the transfer of data and address information in the CPU:

- Microprogram Counter (UPC) bus
- Control Store (CS) bus
- Internal Data (ID) bus
- Memory Data (MD) bus
- Visibility (V) bus
- LSI-11 Data/Address (Q) bus
- Virtual Address (VA) bus and Physical Address (PA) bus
- Synchronous Backplane Interconnect (SBI) bus

3.1.1.1 Microprogram Counter (UPC) Bus - The UPC bus carries addresses from the microsequencer (USC) module that select and fetch microinstructions from the control store ROM or RAM. The microsequencer branch logic tests processor states and conditions and generates the 13-bit addresses that control microprogram flow and sequencing.

3.1.1.2 Control Store (CS) Bus - The CS bus asserts bit fields from the microprocessor control word (microword) to all areas of the CPU The microword consist of 96 data bits and 3 parity bits (1 parity bit for each 32-bit data segment) The microword is divided into thirty bit-fields, each of which controls areas or sections of the CPU logic

3.1.1.3 Internal Data (ID) Bus The ID bus is the high-speed bidirectional data path that handles the following types of information transfers between internal sections of the CPU logic

- Instruction buffer (IB) register transfers to the CPU data paths and the FPA in the form of displacements and short literals
- Data transfers between the CPU data paths and the FPA
- Data transfers between the internal data (ID) registers and other sections of the CPU logic
- Data transfers between the internal data (ID) registers and the console during maintenance procedures

The ID bus consists of 32 data lines, 6 address lines, and 1 write control line The address lines select a register on the ID bus The write control line selects it as a source or destination register, asserting its contents to the ID bus or causing ID bus data to be clocked to it

Table 3-1 lists the ID registers and their address assignments On a normal read, data is clocked from an ID register to the data path Q register On a normal write, data is clocked from the data path D register to the ID register During maintenance functions, the console reads or writes registers on the ID bus and may access the data path Q and D registers as ID registers

ID bus control is derived from the function select (UFS) and console ID (UCID) fields of the microword UFS is a one-bit field If UFS is clear, the ID address and write signals are zero Instruction buffer data is asserted to the ID bus and is clocked to the data path Q register when the Q register is selected.

If UFS is set, the UCID field controls the ID bus, specifying the type of data transfer (read or write) and the address source Table 3-2 lists the address source and operation selected by the UCID field values During normal operation, the internal register addresses are generated in either the shift count (SC) register in the CPU data paths or by the UKMX field of the microword During maintenance, the register address is generated by the console Console control allows access to the ID registers and to the writable control store (WCS).

CNSL ACK notifies the console that the CPU is not using the ID bus and the console may assert its ID maintenance bit and an internal register address CNSL ACK also sets the console command mode flip-flop CNSL CONT then clears the console command mode flip-flop to relinquish control of the ID bus

3.1.1.4 Memory Data (MD) Bus - The MD bus is the bidirectional path for aligned longword data transfers, connecting the CPU data paths and instruction buffer to cache memory and the SBI control.

The MD bus consists of 40 lines. 32 data lines, 4 parity lines, and 4 byte mask lines. The parity lines provide parity for each of the 4 data bytes (parity bit 0 for byte 0, etc.). The mask bits are also associated with the data bytes (mask bit 0 for byte 0, etc.) to inform the system which bytes contain valid data for the transfer

Table 3-1 ID Bus Register Address Assignments

Address (Hex)	Register Name	Address (Hex)	Register Name
00	IBUF DATA	20	USTACK
01	TIME OF DAY	21	UBREAK
02	(Reserved)	22	WCS ADDRESS
03	SYSTEM ID	23	WCS DATA/STATUS
04	CNSL RXCS	24	P0BR
05	CNSL RXDB (TO ID)	25	P1BR
06	CNSL TXCS	26	SBR
07	CNSL TXDB (FROM ID)	27	RSVD FOR SYS SPACE
08	DQ (ID MAINT ONLY)	28	KSP
09	NEXT INTERVAL	29	ESP
0A	CLOCK CS	2A	SSP
0B	INTERVAL COUNTER	2B	USP
0C	CES	2C	ISP
0D	VECT	2D	FPDA
0E	SIR	2E	D.SV
0F	PSL	2F	Q.SV
10	TBUF DATA	30	T0
11	(Reserved)	31	T1
12	TBUF REG 0	32	T2
13	TBUF REG 1	33	T3
14	ACC REG 0	34	T4
15	ACC REG 1	35	T5
16	ACC MAINT	36	T6
17	ACC CONTROL/STATUS	37	T7
18	SBI SILO	38	T8
19	SBI ERR	39	T9
1A	SBI TIMEOUT ADDRESS	3A	PCBB
1B	SBI FAULT/STATUS	3B	SCBB
1C	SBI SILO COMPARATOR	3C	P0LR
1D	MAINTENANCE	3D	P1LR
1E	CACHE PARITY	3E	SLR
1F	CPU CYCLE COUNT	3F	RSVD FOR SYS SPACE

NOTE

Refer to the Translation Buffer, Cache and SBI Control Technical Description (EK-MM78S-TD) for the CPU cycle count register format and bit usage. All other registers are shown in the VAX Maintenance Handbook, VAX-11/780 (EK-VAXV2-HB).

Table 3-2 UCID Field Control of the ID Bus

UCID Field Value	Operation	Address Source
0	NOP	No operation
1	UNUSED	-
2	CNSL ACK	Console
3	CNSL CONT	Console
4	ID DATA ← ID REG	Data paths
5	ID DATA ← ID REG	Microcode
6	ID REG ← ID DATA	Data paths
7	ID REG ← ID DATA	Microcode

Note The UCID field controls the IDBus when the UFS bit is a 1.

Data is transferred on the MD bus under the following conditions

- During interrupt summary read responses to the data paths
- Data requested by the data paths or instruction buffer that is found in cache (a cache hit) is transferred to the data paths or instruction buffer
- Data requested by the data paths or instruction buffer that is not found in cache (a cache miss) is retrieved from main memory The data is simultaneously transferred from the SBI control to cache and to the data paths or instruction buffer
- CPU write data is transferred to the SBI control, and is then asserted on the SBI to be written in memory If the location is valid in cache (a cache hit), the cache data is also updated

3.1.1.5 Visibility (V) Bus - The V bus is used for the diagnostic isolation of CPU or system failures It consists of eight serial data lines (channels), one load enable line, one clock signal line, and one maintenance self-test line (Control signals are generated by the console)

Most CPU modules contain at least one V bus shift register whose data inputs are connected to monitor signals on the module The load signal enables the shift register to parallel load the signals The clock signal then shifts the data serially from the selected channel to a shift register on the console interface board where it can be read by the LSI-11 software Refer to the *VAX-11/785 Console Interface Technical Description* (EK-KC785-TD) for the V bus directory.

3.1.1.6 LSI-11 Data/Address (Q) Bus - The Q bus is the high-speed, bidirectional, asynchronous data path that connects the LSI-11 console subsystem to the CPU through the CIB module Originating in the LSI-11, the Q bus allows the LSI-11 to control other devices in the quad-size LSI-11 backplane (the console terminal and floppy disk) and allows those devices to perform data breaks to LSI-11 memory

Q bus protocol is similar to that of the UNIBUS but uses 16 bidirectional data lines to transfer both address and data information A master device initiates a data transaction by first arbitrating for access to the Q bus for a direct memory access (DMA) transfer, or for a break request (BR) for a program interrupt When the master wins access, it performs an address cycle, asserting the address of a slave with a synchronizing signal When the selected slave responds with a reply signal, the master initiates a data cycle with a signal that also specifies the direction of the transfer

3.1.1.7 Virtual Address (VA) Bus and Physical Address (PA) Bus - With memory management enabled, the address asserted on the VA bus is translated to a physical address from the translation buffer and is asserted on the PA bus to cache memory and the SBI control With memory management disabled, the address on the VA bus is physical and is not translated. The PA bus also asserts a physical address from the SBI control to cache memory for updating or invalidating the cache location.

3.1.1.8 Synchronous Backplane Interconnect (SBI) - The SBI is the bidirectional data path for transfers between the CPU, main memory, and other system devices connected to the CPU backplane The SBI provides parity-checked parallel transfers that are synchronous with the CPU Error checking by the SBI control detects single failures in the information path. However, multiple SBI system failures are not necessarily detected.

Section 1 8 1 of the *Translation Buffer, Cache, and SBI Control Technical Description* (EK-MM785-TD) provides a description of the SBI signal groups.

3.2 CPU MICROPROCESSOR

The CPU microprocessor consists of two sections; the control store (CS) and the microsequencer (USC) Two M7475 joint control store (JCS) modules in slots 18 and 20 make up the CS, providing the PROM control store (PCS) and writable control store (WCS) memory space that stores the CPU microprogram The M7476 microsequencer (USC) module in slot 23 monitors processor states and conditions and provides the control store (CS) addressing that directs the microprogram flow and sequencing.

3.2.1 Joint Control Store (JCS) Module

A "soft" control store allows flexibility in updating the microcode or in implementing engineering change orders (ECOs) Figure 3-1 is a block diagram of the joint control store module which contains the PROM control store in 512 x 99 bits of ROM and the writable control store in 4K x 99 bits of RAM. The two M7475 modules in slots 18 and 20 are configured by a backplane jumper to provide up to 8K of microprogram memory space allocated as shown in Figure 3-2

3.2.1.1 JCS Address Jumper - The module with a floating input on the signal RAM ONLY L disables the upper 5K of RAM, enables 5K of ROM for those addresses and responds to addresses 0000 through 0FFF

The module with a jumper to ground on the signal input RAM ONLY L disables ROM, enables all 4K of RAM and responds to addresses 1000 through 1FFF. The PCS occupies 512 x 99 bit PROM microwords, while the combined WCS of the two modules provides 7 5K x 99 bits of microprogram RAM The backplane jumpering points for JCS module configuration are shown in Table 3-3

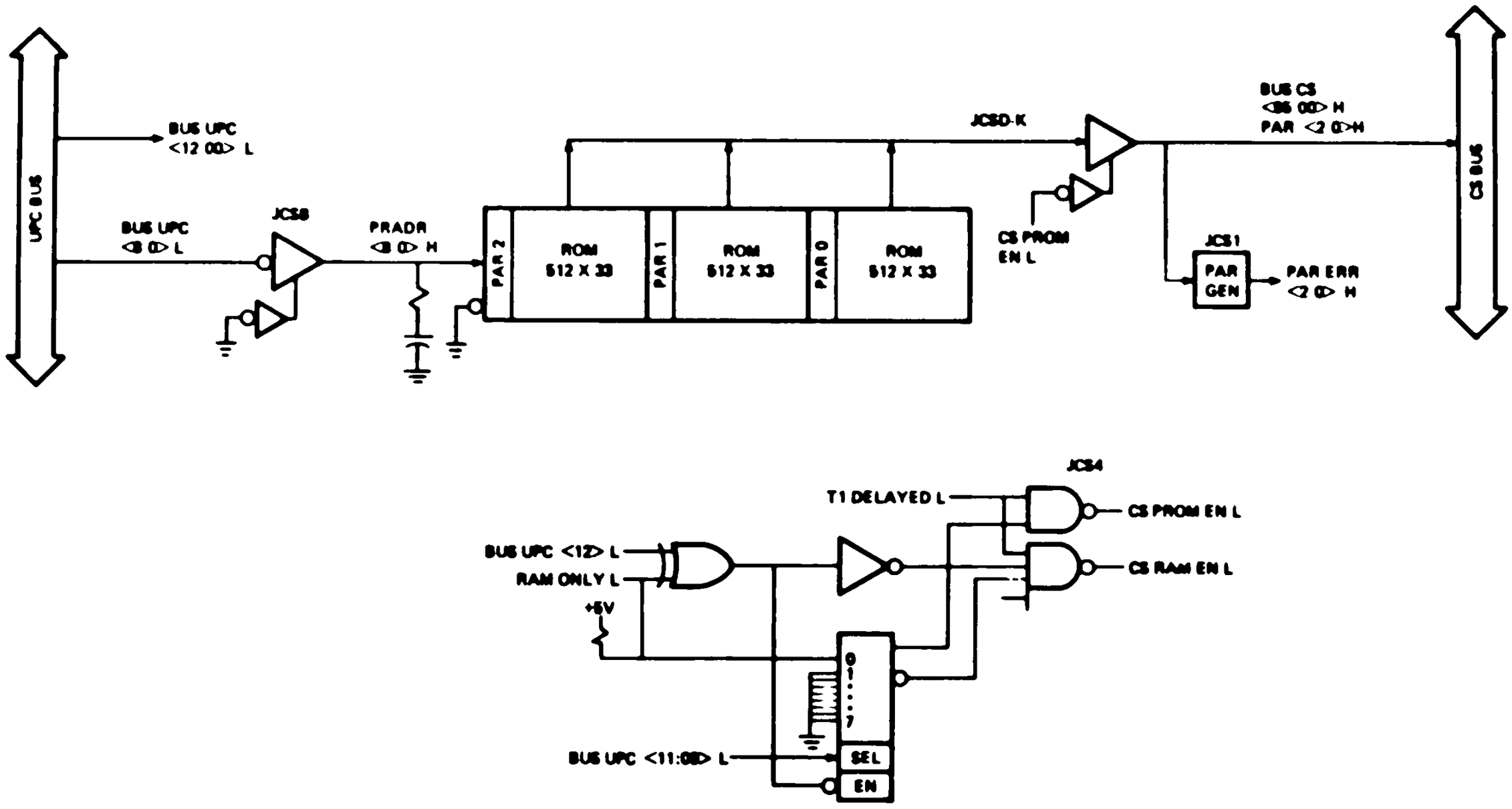
Table 3-3 JCS Module Address Configuration Jumpers

Address Selection		CPU Backplane Jumper Connection		
Slot 18	Slot 20	J11-LL	J11-X	
0 to 4K	5 to 8K	In	Out	(Standard configuration)
5 to 8K	0 to 4K	Out	In	

The PCS contains the microcode for power-up, power down, and microtrap sequences and for some console commands On a power-up sequence, the console subsystem loads the WCS from RX01 floppy disk When the load has completed, the WCS contains all necessary microcode to operate the system, fetch and execute instructions, and perform all memory and I/O functions

3.2.1.2 Control Store Microword and Parity - The microsequencer control word (microword) consists of 96 data bits, 1 spare bit, and 3 parity bits (1 for each 32-bit data segment) WCS parity is generated on and is stored with each 32-bit segment written to the WCS RAM. PCS parity is held in (is part of) ROM.

The JCS uses even parity, with each stored parity bit providing an even number of ones for its 33-bit field If a microword is read from PCS or WCS and the parity checker detects an odd number of ones in any 33-bit field, it asserts a parity error bit (PAR ERR (2:0)) to the interrupt control logic, initiating a microtrap



MLV84 1363

Figure 3-1 Joint Control Store (JCS) Module Block Diagram (Sheet 1 of 2)

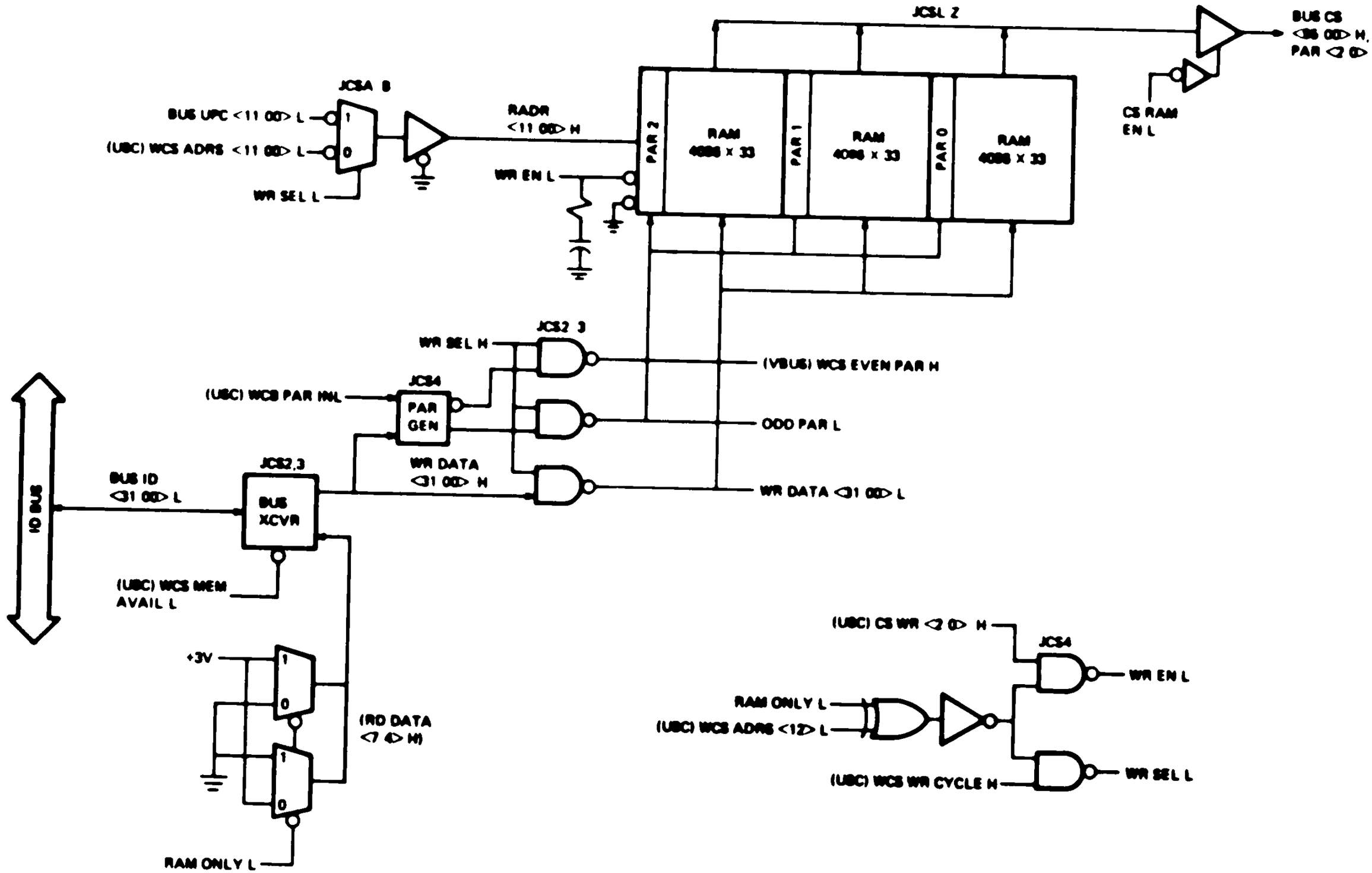


Figure 3-1 Joint Control Store (JCS) Module Block Diagram
(Sheet 2 of 2)

MKV84-1351

3.2.1.3 Control Store Addressing – During normal operation, the microsequencer module asserts microword addresses on the UPC bus to the WCS and PCS sections.

During a power-up sequence or maintenance operations, addresses generated by the console are written on the internal data (ID) bus to the WCS address (WCS ADRS) register in the microsequencer. They are then asserted on the WCS ADRS bus to the JCS. The ID bus is 32 bits wide and data generated by the console is written to the WCS in three 33-bit segments (32 data bits and 1 parity bit per segment).

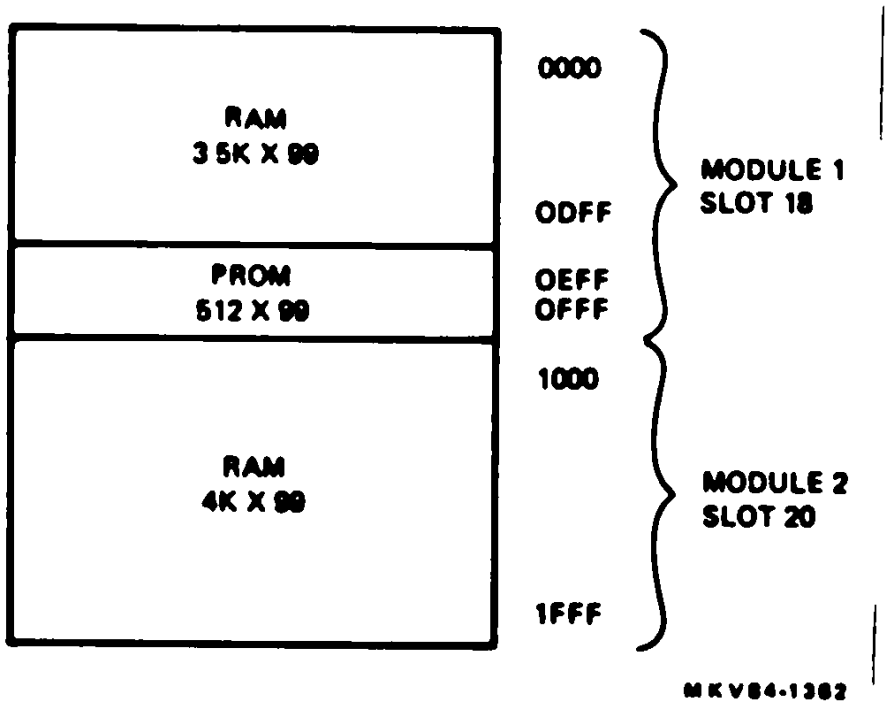


Figure 3-2 Control Store (CS) Address Space

3.2.2 Microsequencer (USC) Module

Figure 3-3 is a diagram of the M7476 microsequencer (USC) module in slot 23. The USC contains the following logic:

- **Microsequencer Mode Control (Picosequencer)** selects the microprogram operating mode according to priority and supplies the mode addressing source to the joint control store on the microprogram counter (UPC) bus.

- **Branch Decoder** provide the next microword address (NUA) to the UPC bus from an address base supplied by the microword on the control store (CS) bus. The branch decodes also provide microprogram sequencing, and select microcode forks and branches, according to processor states.

- **Microstack** stores the current microaddress when a subroutine is called. It restores the current microaddress on a return from the subroutine. When the microcode is in the console wait loop, the console initiates a console command by pushing the breakout address of the microroutine on the microstack, then asserts the MAINT RET signal, which pops the address and forces the microsequencer to enter the console command microroutine.

- **WCS Address and Control** receives address and data information on the ID bus from the console on a power-up sequence or during maintenance. On a power up, the WCS is loaded from RX01 floppy disk under console control.

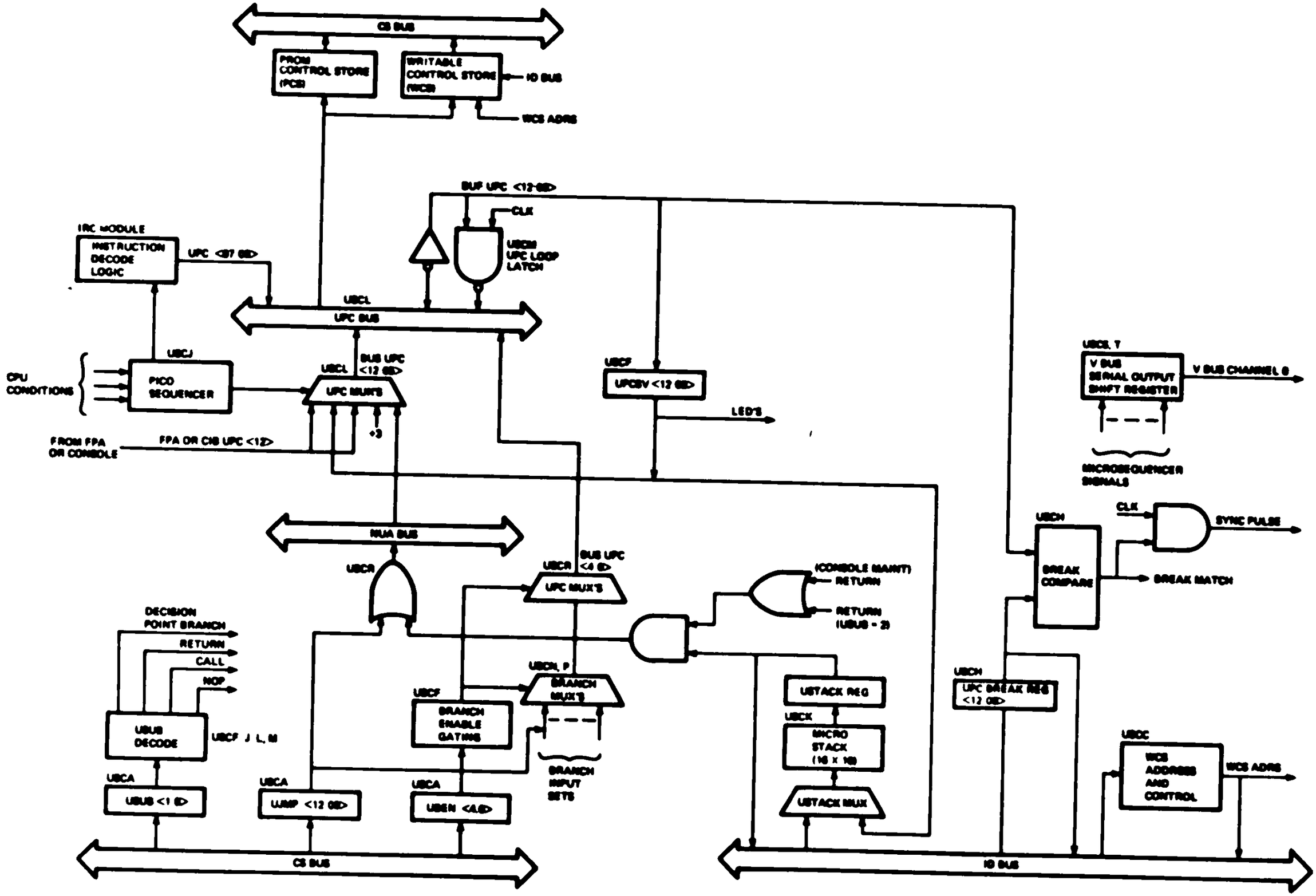


Figure 3-3 Microsequencer (USC) Module Block Diagram

The USC receives and stores three fields from the microword on each microcycle. The jump field is received on BUS CS ⟨12:00⟩ and stored in register UJMP ⟨12:00⟩. The branch enable field from BUS CS ⟨76·72⟩ is stored in UBEN ⟨04:00⟩. The subroutine control field from BUS CS ⟨65:64⟩ is stored in USUB ⟨01:00⟩. These registers are cleared, however, by a cache stall, microtrap, SBI parity error, or a ROM NOP signal from the console.

The UPC loop latch holds each new microaddress for a specific time period to allow the microword parity to be checked. When the UPC multiplexers assert a new microword address on BUS UPC ⟨12:00⟩, the UPC loop latch closes and holds the new microaddress, which is also clocked to the UPCSV register (UPCSV ⟨12:00⟩) The UPC loop latch is closed during the parity check and the UPC MUX outputs are disabled. After the parity check, the latches are opened and the UPC MUX outputs are enabled, allowing the next microword address to be asserted on the UPC bus.

3.2.2.1 Microsequencer Mode Control (Picosequencer) As shown in Figure 3-4, the picosequencer selects the microprogram operating mode according to the following priorities·

Highest: Initialize (INIT) – Power up/Power down
Cache stall (STALL)
Microtrap (UTRAP)

Lowest: Normal

Maintenance return (MAINT RTN) – Console operation on normal console commands or on maintenance functions

The picosequencer selects the address source for the highest asserted priority and asserts it to the joint control store on BUS UPC ⟨12·05⟩. (BUS UPC ⟨04:00⟩ are controlled and asserted by the branch decoder logic)

The operating mode is determined by loading the INIT, STALL, and UTRAP states to a priority decoder that selects the mode for the highest asserted priority. The normal mode is lowest priority. MAINT RTN is a QBUS control bit that is set or cleared by the console and is a separate mode. Priority encoder outputs PRIOR ⟨01·00⟩ assert UMX SEL ⟨01:00⟩, which then select the UPC MUX inputs as shown in Table 3-4, and asserts these inputs to BUS UPC ⟨12:05⟩. (PRIOR ⟨02⟩, asserted, also asserts UMX SEL ⟨01⟩.)

Table 3-4 UPC Multiplexer Input Selection

PRIOR ⟨02:00⟩	UMX SEL ⟨01:00⟩	Selected Mode	UPC Mux Inputs Gated to BUS UPC ⟨12:05⟩
000	00	Normal or console	BUS NUA (12:05)
001	01	(Reserved)	11111111
010	10	Microtrap (UTRAP)	01110000*
011	11	Cache stall (STALL)	UPCSV (12:05)
100	10	Initialize (INIT)	01110000*

* Hardwired address except for BUS UPC ⟨12⟩ which is asserted by the CIB module.

For normal or maintenance mode microsequences, the NUA bus sources the contents of register bits UJMP ⟨12:05⟩ to the UPC bus. An initialize or microtrap gates a hardwired value of 01110000 to BUS UPC ⟨12:05⟩ with bit ⟨12⟩ controlled by the console interface board (CIB). The contents of UPC save register UPCSV ⟨12:05⟩are gated to the UPC bus on a cache stall (STALL).

USUB ⟨01:00⟩ are equal to 0 (NOP) during normal operations. When equal to a 3, for a decision point branch, BUS UPC ⟨07:00⟩are disabled. This allows the instruction decode logic in the IRC module to select those bits for the next microaddress. This also asserts BUS UPC ⟨12⟩ with an inclusive-OR of BUS NUA ⟨12⟩ and ACC OVERIDE from the FPA. (The other USUB values call or return from a microsubroutine and affect the microstack as described in Section 3.2.2.7.)

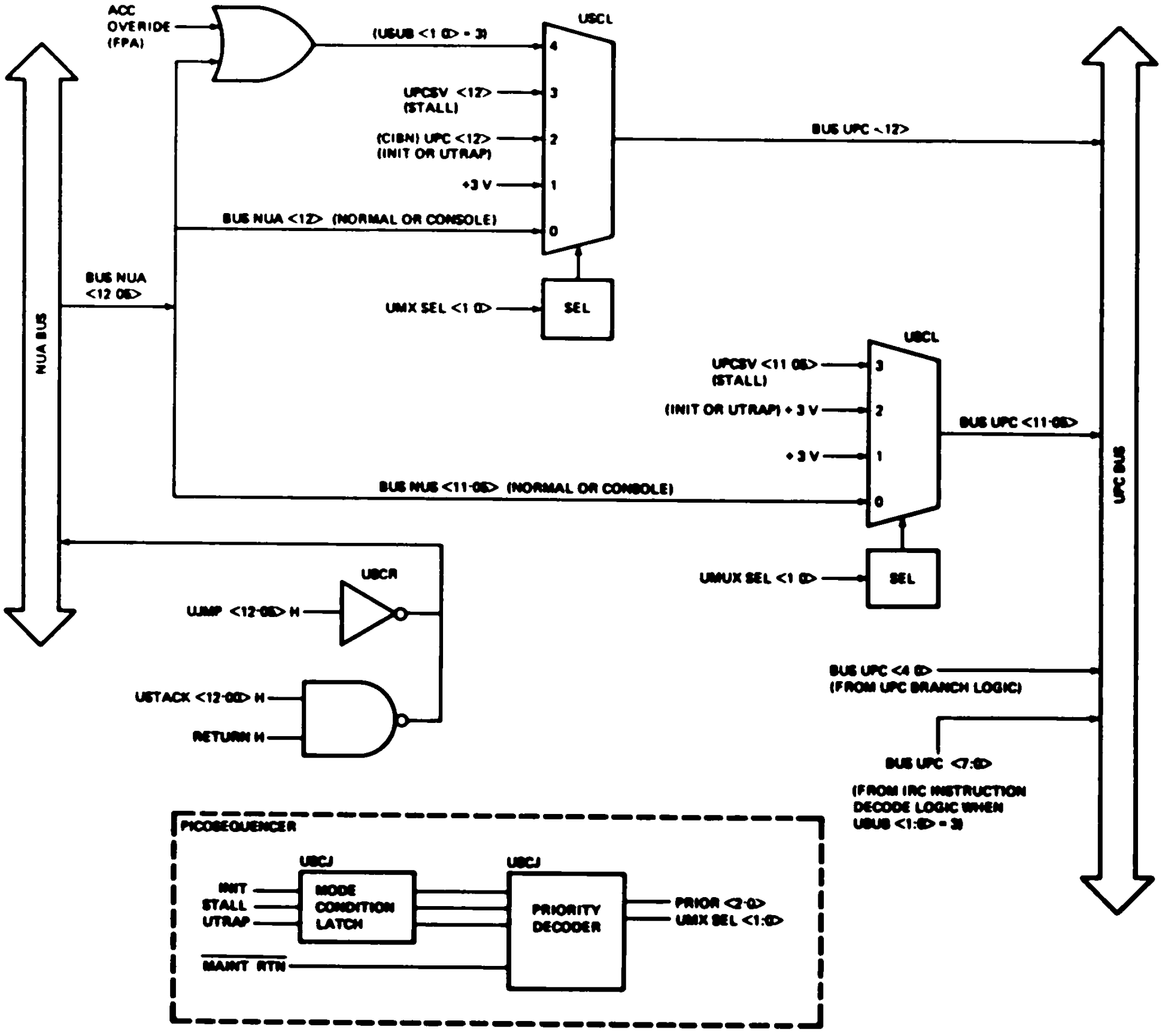


Figure 3-4 Picosequencer and Mode Multiplexers

3.2.2.2 **Branch Decoder** - Figure 3-5 is a block diagram of the multiplexers that monitor the processor registers and states which control microprogram sequencing When enabled, the following three multiplexer groups assert BUS UPC (04 00) on the UPC bus where they are concatenated with BUS UPC (12:05) from the picosequencer

- NUA MUX - Next microaddress multiplexer
- BR MUX (F:0) - Branch multiplexer set (F:0)
- BR MUX (1F:10) - Branch multiplexer set (1F 10)

During normal microsequences, the NUA MUX asserts the contents of register UJMP (04:00) to the UPC bus There, they are inclusive-ORed with a BR MUX to branch on conditions tested by the UBEN field of the microword On a cache stall (STALL), the NUA MUX asserts the contents of UPCSV (04:00) On a return from subroutine (RETURN) or a maintenance return (MAINT RTN) from the console, the NUA MUX asserts an inclusive-OR of USTACK (04:00) from the microstack with UJMP (04.00)

Table 3-5 shows the branch test conditions selected by UBEN (04.00). UBEN (04) on a 0 enables the BR MUX (F:0) multiplexers that test the processor registers and conditions shown in Table 3-6. UBEN (04) on a 1 enables the BR MUX (1F.10) multiplexers that test the registers and conditions shown in Table 3-7. Note in these tables that, while UBEN (03:02) select the BR MUX inputs to the UPC BUS, UBEN (02:00) further select the inputs of three other multiplexers (BR BIT and BRMX).

Table 3-5 L BEN Field Control of the Microcode Branch Condition Sets

L BEN (04:00) Value	Branch Set Selected	UBEN (04:00) Value	Branch Set Selected
0	NOP	10	UTRAP vector
1	Z	11	Last reference
2	ROR	12	EALU CC
3	ALU C31	13	(Unused Code)
4	BC4 (02:00)	14	SC
5	B0 valid	15	ALU (01:00)
6	FPA	16	State (07:04)
7	(Unused Code)	17	State (03:00)
8	Data type (VAX)	18	D bytes
8	END DPI (PDP-11)	19	D (03:00)
9	IR (02:01) (VAX)	1A	PSL CC
9	PC modes (PDP-11)	1B	ALU CC
		1C	PSL Mode
A	REI	1D	Translation Test
B	IB test	1E	(Unused Code)
C	MUL	1F	(Unused Code)
D	Signs		
E	Interrupt		
F	Decimal		

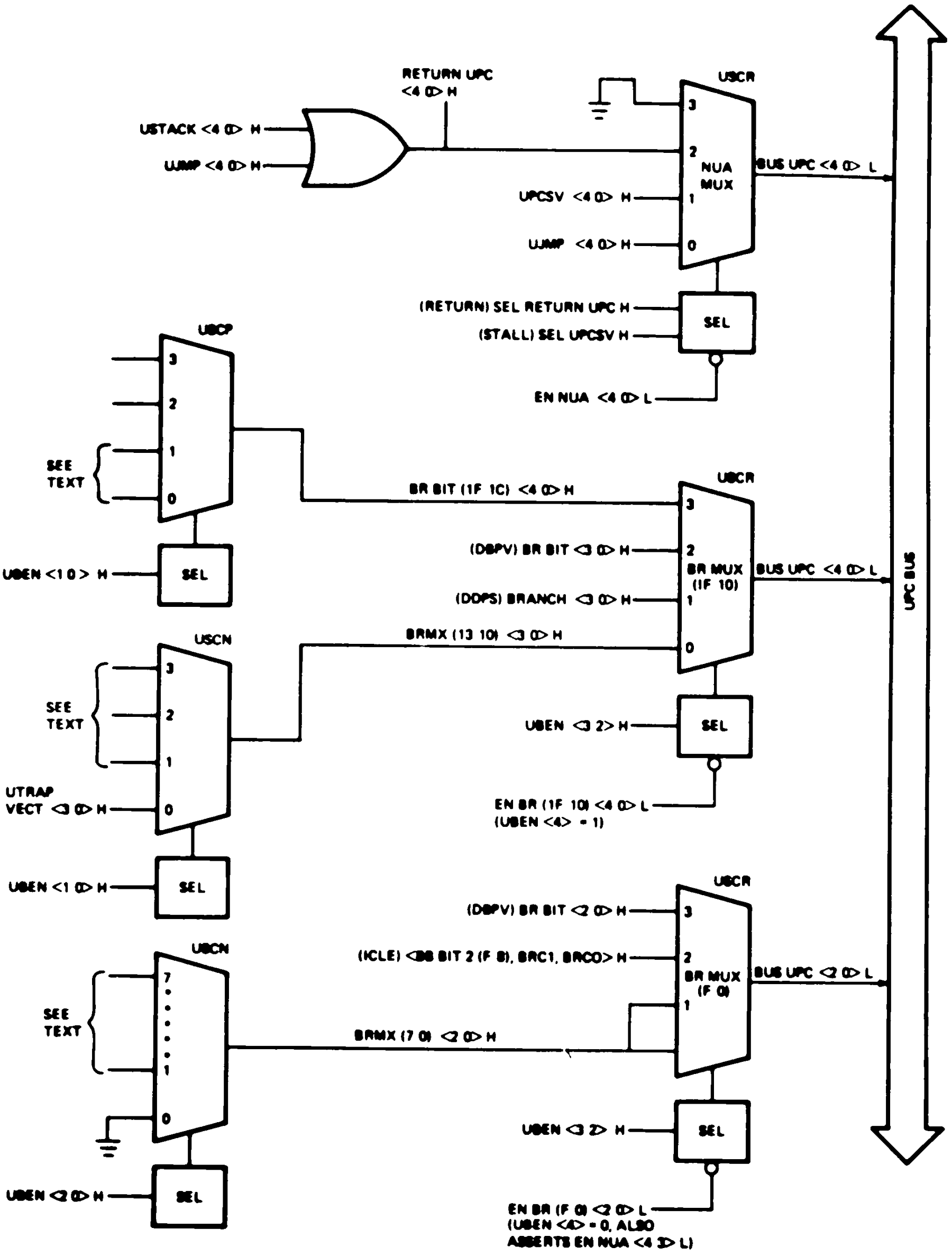


Figure 3-5 Branch Decode Multiplexers

Table 3-6 BR MUX (F:0) Input Selection for BUS UPC (02:00)

UBEN (04:00)	Inputs from the BRMX (7:0) (2:0)Multiplexer Bit (02)	Bit (01)	Bit (00)
0	0	0	0
1	0	0	ALU Z
2	LA01	PSL CBIT	LA00
3	0	ALU C31	0
4	BC4 (2)	BC4 (1)	BC4 (0)
5	B0 VAL	0	0
6 (FPA)	UB2	UB1	UB0
7*	0	0	0
UBEN (04:00)	**Inputs from the BCL and DBP Modules Bit (02)**	**Bit (01)**	**Bit (00)**
8-B	BR BIT2 (B:8)	BRC1 (B:8)	BRC0 (B:8)
C-F	BR BIT (2)	BR BIT (1)	BR BIT (0)

* unused code

Table 3-7 BR MUX (1F:10) Input Selection for BUS UPC (4:0)

UBEN (04:00)	Inputs from the BRMX (13:10) (3:0)Multiplexer Bit (04)	Bit (03)	Bit (02)	Bit (01)	Bit (00)
10	-	UTRAP VECT (03)	UTRAP VECT (02)	UTRAP VECT (01)	UTRAP VECT (00)
11	-	$\overline{\text{FPD BIT}}$	NESTED ERR	LST REF CD (01)	LST REF CD (00)
12	-	EALU N	EALU Z	SC NE (0)	SS
13*	-	0	0	0	0

* unused code

Table 3-7 BR MUX (1F:10) Input Selection for BUS UPC (4:0) (Cont)

UBEN (04:00)	Inputs from the BRMX (13:10) (3:0)Multiplexer Bit (04)	Bit (03)	Bit (02)	Bit (01)	Bit (00)
14-17	-	BRANCH (3)	BRANCH (2)	BRANCH (1)	BRANCH (0)
18-1B	-	BR BIT (3)	BR BIT (2)	BR BIT (1)	BR BIT (0)
UBEN (04:00)	**Inputs from the BR BIT (1F:1C) (4:0)Multiplexer Bit (04)**	**Bit (03)**	**Bit (02)**	**Bit (01)**	**Bit (00)**
1C	VAMX (31)	VAMX (30)	CNSL CMD M	$\overline{\text{IS+CM}}$	KERNEL MODE
1D	PTE INVALID	DATA ALIGNED	0	BR CODE (1)	BR CODE (0)
1E*	0	0	0	0	0
1F*	0	0	0	0	0

* unused codes

3.2.2.3 **Branch Multiplexer Enable Gating** - Figure 3-6 shows the gating that enables the UPC multiplexer outputs When selected, each of the following control signals enables an individual output bit of the multiplexer

- EN NUA (04:00) - Enable NUA MUX (04:00) bits
- EN BR (1F 10) (04 00) - Enable BR MUX (1F 10) (04 00) bits
- EN BR (F:0) (02:00) - Enable BR MUX (F 0) (02 00) bits

Enable signals are asserted from the rising edge of clock T33 to the rising edge of clock T99 All outputs are disabled when USUB (01:00) equal 3 (decision point branch) allowing the IRC module to assert UPC BUS bits (07:00) from the instruction decode logic

UBEN (04) from the microword affects branch multiplexer selection and the general operation of this logic as follows

1 UBEN (04) on a 0 asserts UBEN EN (F:0), enabling the BUS UPC (04:00) outputs for BR MUX (F:0) On a 1, it asserts UBEN EN (1F 10), enabling the BUS UPC (04:00) outputs for BR MUX (1F 10)

During normal microsequences, 1s in UJMP (04:00) assert themselves on the UPC BUS from the NUA MUX and disable the corresponding bits from the BR MUXes. The 0s in UJMP (04:00), however, disable the corresponding outputs from the NUA MUX and enable them for the corresponding BR MUX

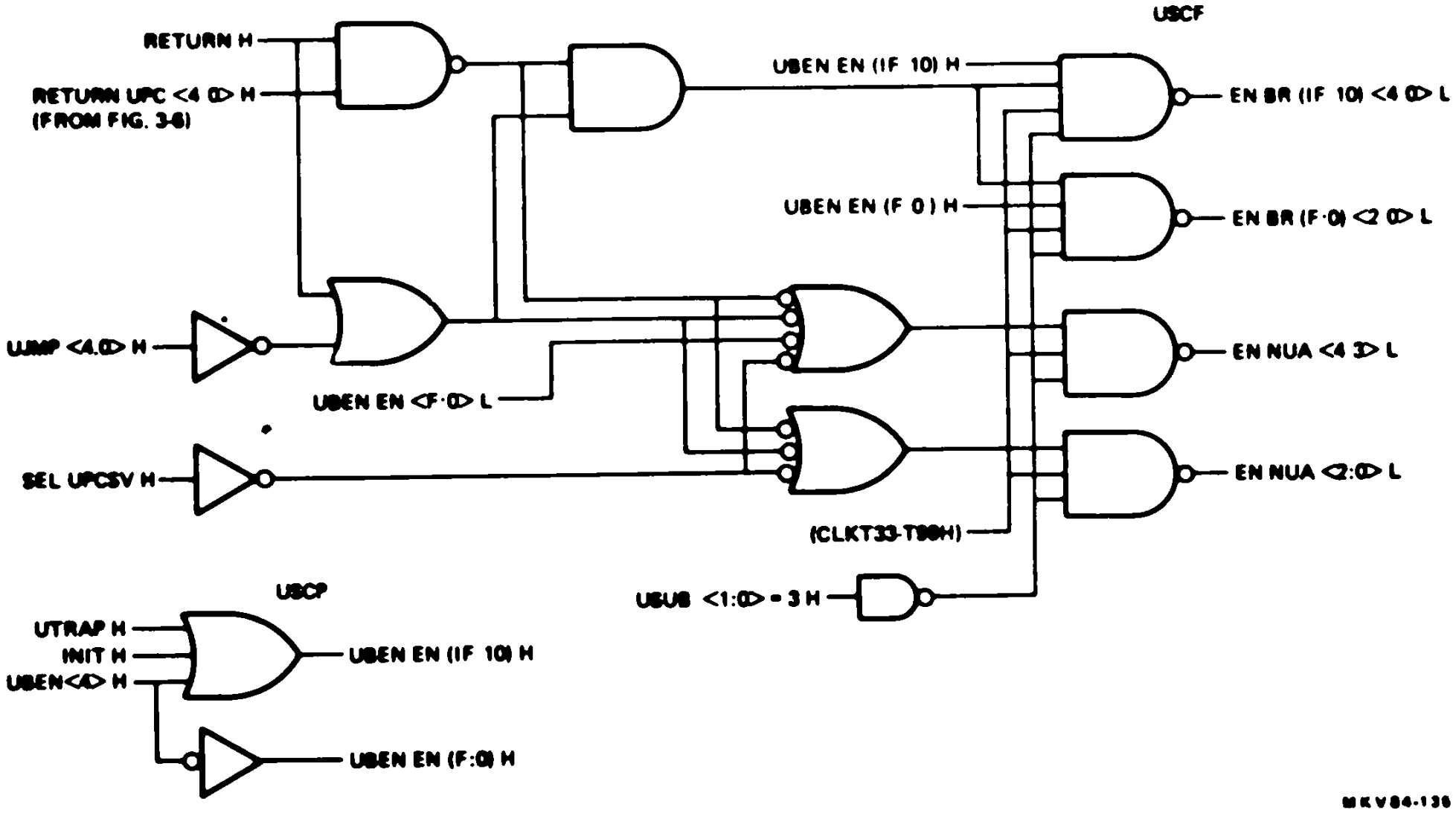


Figure 3-6 Branch Multiplexer Enable Gating

2 UBEN EN (F:0), when asserted, also asserts EN NUA (04:03) This causes bits (04:03) from the NUA MUX to be inclusive-ORed on the UPC BUS with bits (04:03) from BR MUX (F:0)

3 UBEN EN (1F 10) is also asserted on an initialize or microtrap, allowing the power up or microtrap vector to be asserted on the UPC BUS

4 On a STALL, SEL UPCSV enables all NUA MUX bits The NUA MUX asserts contents of UPCSV (04:00) to the UPC BUS where they are concatenated with the contents of UPCSV (12:05) from the processequencer when leaving the stalled condition

5 On a RETURN from subroutine (USUB (01:00) = 2), the inclusive-ORed contents of USTACK (04:00) and UJMP (04:00) (the resulting 1s) are asserted from the NUA MUX on the UPC BUS The resulting 0s enable the corresponding BR MUXes

3.2.2.4 Initialize (INIT) Mode - Powering the system up or down generates the INIT signal, forcing the microsequencer to place microtrap vector 0E:00 on the UPC bus This clears all microsequencer registers except for the VBUS, UPC save, and UPC break registers, allowing the console to read them for maintenance purposes

The microsequencer remains in the INIT mode for one microcycle after the system INIT level is negated when power is stable The microjump (UJMP (12:00)) and microbranch enable (UBEN (04:00)) fields of the microinstruction then assert the address for a microsequencer start-up in the normal mode

3.2.2.5 Normal Mode - Under normal microsequencing, UJMP (12:05) are sourced from the NUA bus to the UPC bus, while UJMP (04:00) are directly asserted to the UPC bus to select the next microword address The value held by UBEN (04:00) tests CPU conditions that may modify any BUS UPC (04:00) bits not asserted as 1s by UJMP (04:00) (an inclusive-OR).

3.2.2.6 Cache Stall (STALL) Mode - STALL mode is initiated when the microsequencer receives SBLT STALL from the SBI control In this mode, the execution of the next microinstruction is prevented and the signal CLEAR UWORD is generated This places the microprogram in a NOP state that may last for several microcycles until the condition causing the stall is negated

When the stall condition is negated, the UPC MUX selects the UPC save register UPCSV (12:00) as the source for the next microaddress The UPC save register contains the address of the next instruction that would have been executed if the stall had not occurred The USC module provides 13 LEDs that display the contents of the UPC save register and one LED that displays the STALL state.

3.2.2.7 Microstack Operation Figure 3-7 is a diagram of the microstack which stores up to sixteen 16-bit microaddresses, and its addressing and control logic

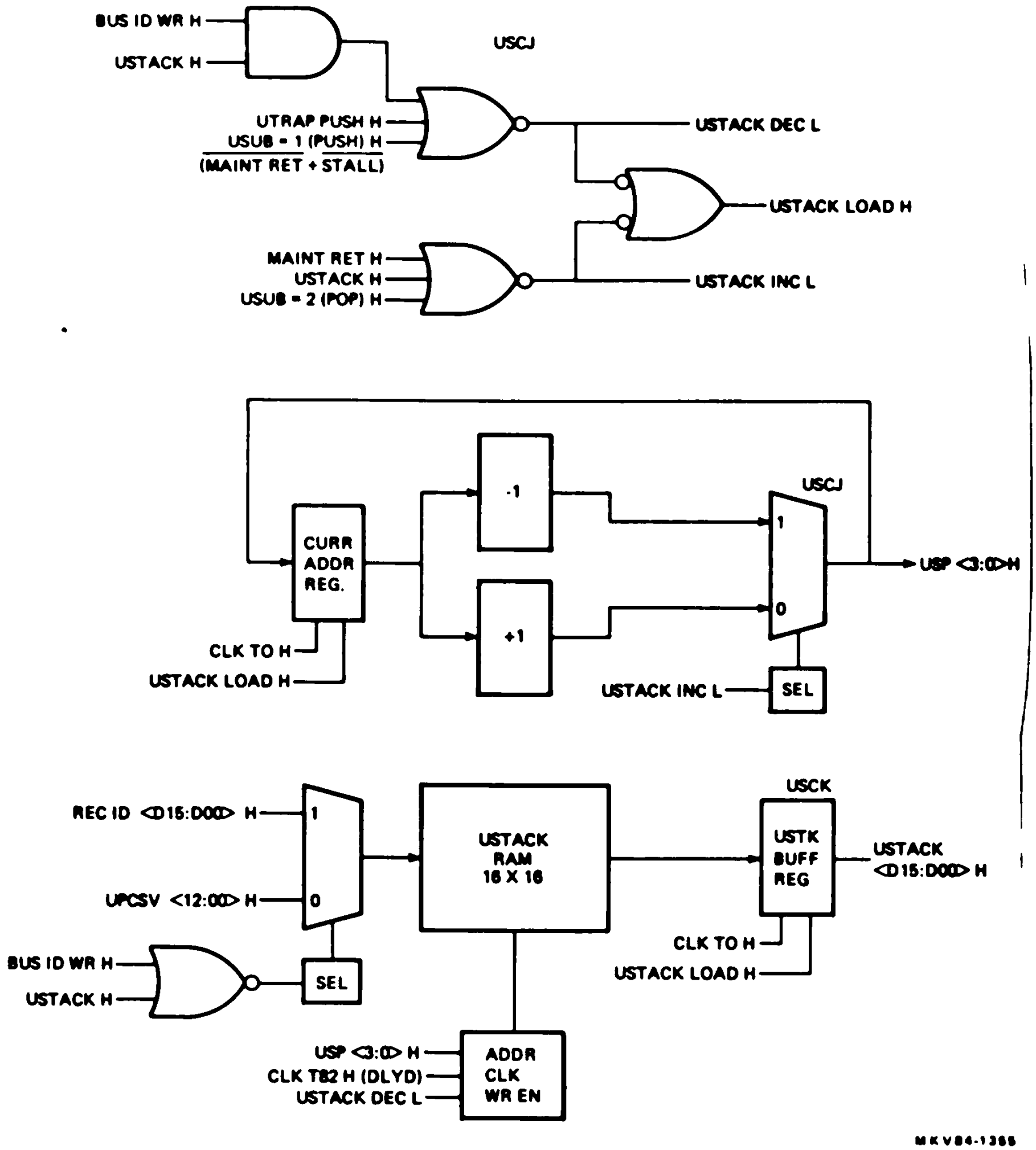


Figure 3-7 The Microstack (USTACK)

3.2.2.7.1 **Subroutine Control** - For other than normal operation, the USUB ⟨01:00⟩ field of the microword specifies a decision point branch or the call or return from subroutine operation as shown in Table 3-8.

Table 3-8 USUB Field Control Functions

USUB ⟨01:00⟩	Function
00	NOP (normal operation)
01	Call subroutine (CALL)
10	Return from subroutine (RETURN)
11	Decision point branch

When a CALL subroutine is specified, the microstack pointer (USP ⟨03:00⟩) is first incremented. The address of the current microinstruction, held by UPCSV ⟨12:00⟩, is then pushed onto the microstack to be used in forming the return microaddress.

When a RETURN from subroutine is specified, the contents of UPCSV ⟨12:00⟩ are popped from the microstack and inclusive-ORed with the UJMP ⟨12:00⟩ field. The result of the OR function points to the address of the microinstruction that followed the CALL instruction.

3.2.2.7.2 Microstack Addressing – The outputs of the current address register are always asserted to the plus-one (incrementing) adder and minus-one (decrementing) adder. The deasserted state of USTACK INC L from the control gating normally asserts the outputs of the minus-one adder to the outputs of the microstack pointer multiplexer (USP ⟨03:00⟩).

3.2.2.7.3 Microstack Push – A write occurs to the microstack under one of the following conditions:

- When the USUB ⟨01:00⟩ field is equal to 1 (CALL)
- On a UTRAP condition (if STALL is deasserted)
- On a maintenance mode ID bus write to the microstack

Each of the above conditions, if active at the beginning of a CPU cycle, asserts the microstack decrement (USTACK DEC) and microstack load (USTACK LOAD) signals. The UTRAP or subroutine CALL condition pushes the contents of the UPC save register (UPCSV ⟨12:00⟩) on the microstack. Maintenance mode allows control store addressing by pushing a microaddress on the microstack from the ID bus (BUS ID ⟨15:00⟩).

With USTACK INC deasserted, the decremented contents of the current address register are available on USP ⟨03:00⟩. USTACK DEC enables the write inputs to the microstack and the data is clocked to the microstack at T82 of the same CPU cycle. USTACK LOAD enables the inputs to the current address register and to the microstack output buffer register (USTACK ⟨D15:D00⟩). The following events then occur at CLK T0 of the next CPU cycle:

- The decremented contents of the current address register are clocked back to the current address register.
- The data just written to the microstack is clocked to the microstack output buffer register where it is available for the next pop operation.

3.2.2.7.4 Microstack Pop – The microstack is then read under one of the following conditions:

- When the USUB ⟨01:00⟩ field is equal to 2 (RETURN)
- When maintenance return (MAINT RTN) is set by the console

Initially, the last word written to the microstack and currently held by the microstack output buffer register is used as the popped address for the return. Then, each of the above conditions also asserts the microstack increment (USTACK INC) and microstack load (USTACK LOAD) signals.

With USTACK INC asserted, the incremented contents of the current address register are available on USP ⟨03:00⟩. USTACK LOAD enables the inputs to the microstack output buffer register and the inputs to the current address register. CLK T0 of the next CPU cycle produces the following events:

- The microstack data (currently selected by the incremented contents of the current address register) is clocked to the microstack output buffer register where it is available for the next pop operation.
- The incremented contents of the current address register are clocked back to the current address register

3.2.2.8 Microtrap (UTRAP) Mode - If no higher mode condition is present, the microsequencer enters the UTRAP mode and generates a vector address that accesses the trap handling subroutine in the PCS

On receipt of the UTRAP signal, the microsequencer asserts the ABORT CYCLE signal to the TBM module It asserts a hardwired value of 0E0 on BUS UPC (12:05) (BUS UPC (12) comes from the console), and sources BUS UPC (04:00) from the microbranch logic BUS UPC (03:00) are asserted with the values of UTRAP VECT (03:00) from the CEH module, forming the vector address for the following detected conditions

Table 3-9 shows the vector address ranges that are contained in PROM.

Table 3-9 PROM Vector Address Ranges

Vector Addresses	Routine Entry Points
0E00–0E0F	Machine UTRAP service routines
0F20–0F2F	Console service routines
0FF0	HALT loop
0FFE	Microsequencer error (not supposed to get here)
0FFF	PCS version number

Table 3-10 shows the machine UTRAP service routines that are called from entries listed in Table 3-9.

Table 3-10 Conditions Causing Machine Microtraps

Vector Address	Machine Microtrap Condition
0E00	Machine initialize and start
0E01	Unaligned data
0E02	Page boundary
0E03	Set TB modify (M) bit
0E04	Protection violation
0E05	TB miss
0E06	Reserved operand
0E07	TB parity error
0E08	Cache parity error
0E0C	Read data substitute (RDS) error
0E0D	Read timeout
0E0E	Odd address error (compatibility mode)
0E0F	CS parity error

Multiple UTRAP conditions can be present at the same time. They are, therefore, output from a priority decoder in the CEH module that determines which microtrap will be serviced first. Their priorities are as follows:

Machine initialize and start
CS parity error
Odd address error (compatibility mode)
Read timeout
Read data substitute (RDS) error
Cache parity error
TB parity error
Reserved operand
TB miss
Protection violation
Set TB modify (M) bit
Page boundary
Unaligned data

When a microtrap is initiated, the contents of the UPC save register are pushed onto the microstack. The contents of the UPC save register specify the location of the next microword that would have been selected under normal conditions. It is saved so that the microprogram can return to normal flow after the microtrap has been serviced

The control store parity error microtrap is one exception to storing the UPC save register contents on the microstack. In this case, UPC save register clocking is inhibited to prevent loading it with the current microaddress. The failing microword address is then pushed on the microstack, rather than the address of the next executable microword.

3.2.2.9 WCS Addressing - Figure 3-8 shows the WCS addressing logic and decoder that operate under the direction of the console. When ID LEFT ADDR (05:00) equals 22 with ID LEFT WRITE asserted, the WCS address register receives the microcode starting address from the ID bus. When ID LEFT ADDR (05:00) equals 23 with ID LEFT WRITE asserted, the write control signals are enabled to the WCS. WCS MEM AVAIL asserted to the WCS enables the data inputs to receive microcode data from the console on the ID bus.

The console writes the starting address to the WCS ADRS counter register on the ID BUS. Address bits WCS ADRS (12:00) are asserted to the control store and are incremented as each location is written. As WCS ADRS (14:13) are incremented, they assert the signals shown in Table 3-11. These signals select each 33-bit segment for writing.

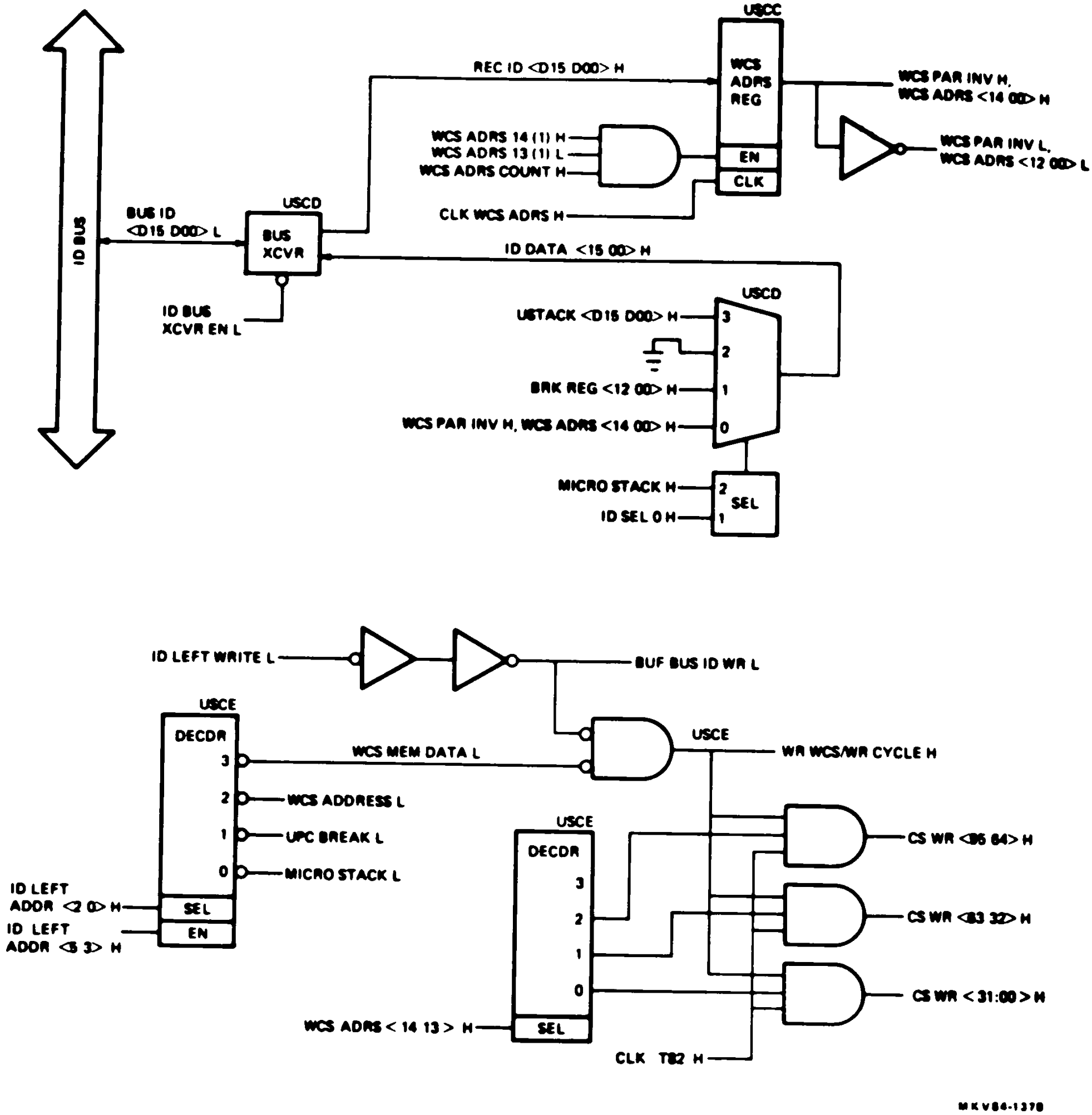


Figure 3-8 WCS Addressing Logic

Table 3-11 WCS Address Selection on a WCS Lead

WCS ADRS Bits (14)	(13)	Signal to WCS	Microcode Segment Written
0	0	CS WR (31:00)	Data bits (31:00)
0	1	CS WR (63.32)	Data bits (63.32)
1	0	CS WR (95 64)	Data bits (95:64)
1	1	Inhibits further writes or address increments	

Parity invert bit (15) (WCS PAR INV) is not part of the WCS ADRS register, but is used for diagnostic purposes to generate odd parity and force a parity error microtrap.

WCS WR CYCLE disables the CS drivers in the WCS to turn off the control store outputs during the write operation The console writes WCS starting with the lower segment, writing one segment at a time in sequential addresses, and proceeding through the entire control store address space. (ROM is not written, although writing may continue through ROM address space.)

The ID bus is divided in half to reduce loading on the ID bus lines by using separate drivers ID registers are then selected on either the right or left half of the ID BUS control The microsequencer is on the left half of the ID bus Its registers are addressed by ID LEFT ADDR (05:00) with data flow direction controlled by ID LEFT WRITE.

On a power-up sequence or maintenance mode, the console asserts an ID bus address and the write control line read or write state The address decoder then selects the microprocessor ID registers shown in Table 3-12

Table 3-12 ID Register Selection on a Power-Up or Maintenance Operation

ID Register Address	Microprocessor Register Selected
20	The microstack (USTACK (D15:D00))
21	The UPC break register (BRK REG (12:00))
22	The WCS address register (WCS ADRS (14:00) and WCS PAR INV)
23	The data inputs to WCS RAM (BUS ID (31:00))

3.2.2.10 Maintenance Mode - In maintenance mode, the console may read or write all four ID registers in the microsequencer

With the processor clock stopped, the console may specify a WCS microaddress by writing it to the microstack When ID LEFT ADDR (05:00) equals 20 with ID LEFT WRITE asserted, the USTACK multiplexer receives the data on REC ID (15:00) from the ID bus The microstack pointer selects a microstack address and the ID bus data (the WCS microaddress specified by the console) is written to the stack When the console asserts MAINT RET, the microstack data is enabled on the NUA BUS to the UPC MUX where it is used as the address for the next microinstruction. The console then proceeds with single step functions

The UPC break register may be read or written with ID LEFT ADDR (05:00) equal to 21 With ID LEFT WRITE asserted, ID bus data on REC ID (15:00) is clocked to the break register The output of the break register (BRK REG (12:00)) is connected to a network that compares it with BUF UPC (12:00) (the address of the next microinstruction to be performed) When the address in the break register matches the address of the next microinstruction, the comparator generates the break match (BRK MAT) signal which is available on a backplane pin as the signal SYNC PULSE SYNC PULSE can be used as an oscilloscope syncpoint on any UPC address specified by the break register

The console may stop the processor clock at a specific microaddress by writing the microaddress to the break register and setting the CLK STOP bit in the console. The console can also force the microsequencer into a NOP cycle by asserting the ROM NOP signal. This signal clears the microword bit field registers and asserts the ABORT CYCLE signal to the TBM module

The V bus serial output shift register is used for maintenance purposes A number of microsequencer signals are first parallel loaded into the V bus register and then shifted out serially to the console The V bus has eight selectable serial channels. Channel 0 is used by the microsequencer, allowing the observation of numerous microsequencer conditions when the processor clock is stopped.

3.3 INSTRUCTION BUFFER AND DECODER

The instruction buffer and decoder logic consists of two modules; the instruction data path (IDP) module in slot 7, and the instruction decoder (IRC) module in slot 8

The IDP contains an eight-byte instruction buffer register that allows a prefetch of instruction stream (I-stream) data from memory It does this by receiving and storing new instruction information while the current instruction is being executed On a fetch, the IDP receives I-stream data from the memory data (MD) bus and clocks it to the lower bytes of the buffer where it is evaluated while further data is fetched from memory

The IRC receives and decodes bytes 0 and 1 of the current instruction from the IDP instruction buffer (instruction opcode and first operand specifier) It then uses this information to direct CPU microcode flow when the CPU microcode reaches a decision point branch An instruction that consists of one opcode, such as a NOP, executes immediately because no operand specifier must be evaluated (In this case, byte 1 would contain the opcode of the next instruction.)

An instruction opcode indicates how many specifier evaluations must be performed. Each specifier indicates how much literal or displacement data follows. The register bytes are then shifted as literal and displacement data is loaded to the internal data (ID) bus as the instruction executes (The moving of register bytes is controlled by the UIBC field of the CPU microword.)

3.3.1 Instruction Data Path (IDP) Module

Figure 3-9 is a block diagram of the IDP module that contains the following sections of logic:

- Buffer register (BUF (B7 B0)) and input multiplexer (IMX)
- Memory data (MD) byte shifter
- Shift multiplexer (SHF MUX)
- Data multiplexer (DMX)

3.3.1.1 Instruction Buffer (IB) Register - The instruction buffer register holds eight 9-bit bytes of I-stream data ((B7.B0)) Each byte consists of eight data bits and one validity bit that is set when the byte is loaded with valid I-stream data.

On a native mode fetch, the instruction op code is loaded to byte 0 for decoding and the operand specifier is loaded to byte 1 for evaluation Bytes 2 through 7 may then contain succeeding I-stream data (an operand or operand address, literal or displacement data for the specifier in byte 1, the next operand specifier, op code, etc)

On a compatibility mode fetch, a 16-bit PDP-11 instruction is loaded to bytes 0 and 1. Bytes 2 and 3 may then contain literal data for the current instruction (index mode information, for example) or may contain the next instruction Bytes 4 through 7 are then loaded with the next instructions while the current instruction is being executed

The instruction buffer also transfers displacement or literal data to other sections of the CPU on the internal data (ID) bus The data multiplexer (DMX) selects the buffer register bytes to be transferred and can sign-extend or shift the data it asserts on the bus

3.3.1.2 Instruction Buffer Addressing - During opcode and operand specifier evaluation, the instruction buffer supplies the following addresses for the scratchpad register sets in the CPU data paths as as shown in Figure 3-10:

- SP1 ADR (03:00) — Specifier 1 (SP1) register address
- SP2 ADR (03:00) — Specifier 2 (SP2) register address
- PRN ADR (03:00) — Previous register number address latch

In native mode, SP1 is the register number from the operand specifier (source register) being evaluated in byte 1 of the buffer register as shown in Figure 3-11 If the operation is a short literal-to-register transfer, or a register to register transfer. SP2 is the register number from the operand specifier (destination register) in byte 2 of the buffer register The SP1 register number is also stored in the PRN latch for use as the scratchpad address source

In compatibility mode, the value of SP1 is determined by the source register field of the instruction and SP2 is determined by the destination register field as shown in Figure 3-12.

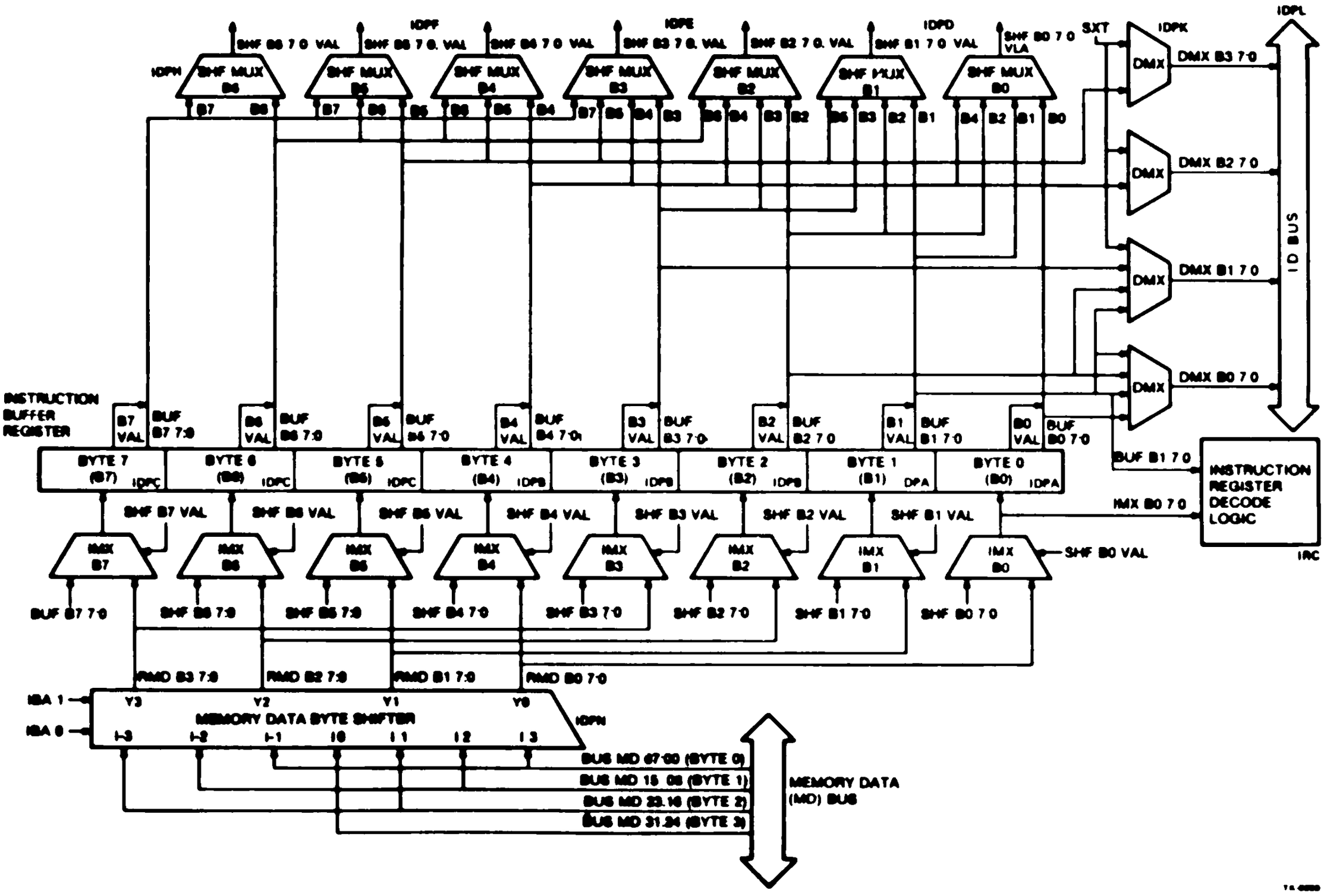


Figure 3-9 Instruction Data Path (IDP) Module Block Diagram

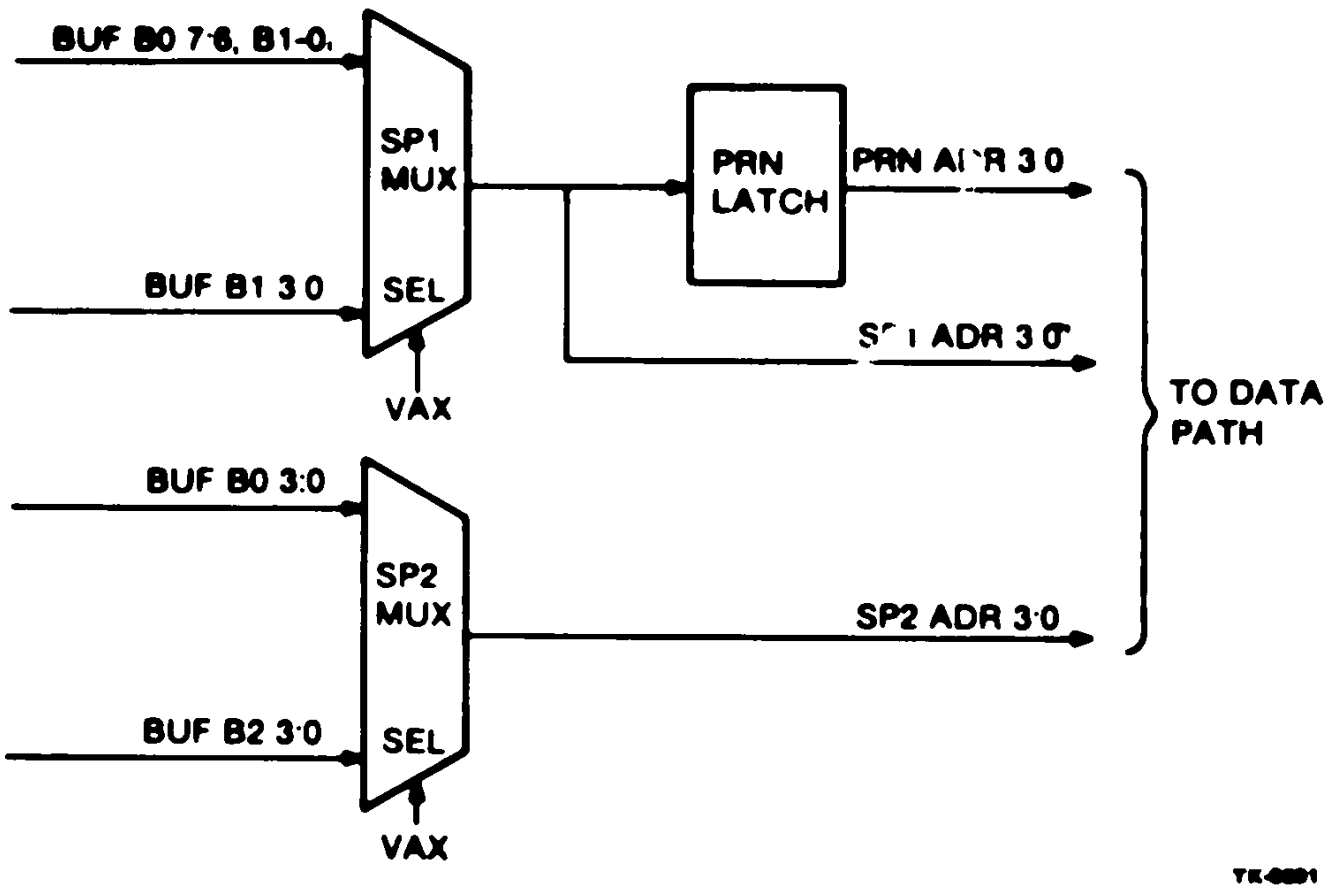


Figure 3-10 Register Addressing Multiplexers

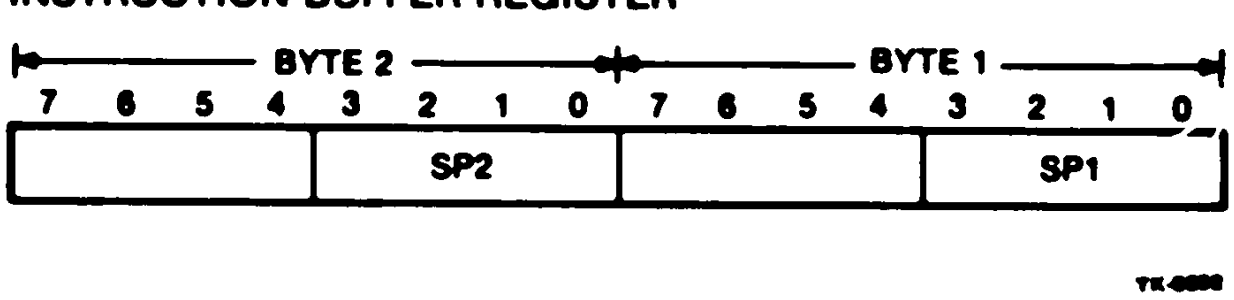


Figure 3-11 Instruction Register Fields in VAX Instructions

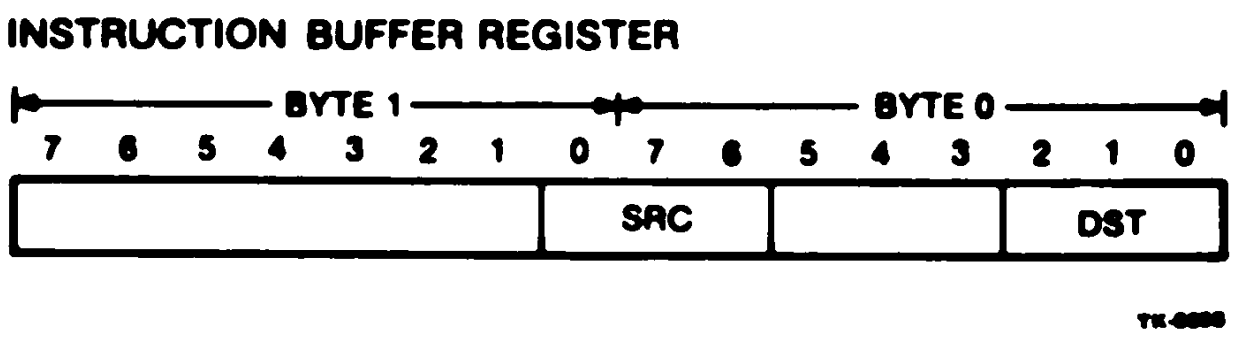


Figure 3-12 Instruction Register Fields in PDP-11 Instructions

3.3.1.3 **Program Counter (PC) Updates** The program counter in the CPU data paths holds the address of the instruction opcode at the start of each new instruction execution sequence As operand specifiers are evaluated, the instruction buffer logic generates a three-bit number (DELTA PC (02:00)) that is added to the low-order bits of the PC register, causing the PC to point beyond the instruction byte being evaluated

In native mode, the PC update value reflects the current operand specifier and any additional bytes of literal or displacement data associated with it The addressing modes shown in Table 3-13 require additional data and also require that the length number be added to the PC

Table 3-13 Program Counter Updates

Addressing Mode	Length Number
Autoincrement (R = PC	1, 2, or 4 bytes
Autoincrement deferred (R = PC)	4 bytes
Byte displacement	1 byte
Byte displacement deferred	1 byte
Word displacement	2 bytes
Word displacement deferred	2 bytes
Longword displacement	4 bytes
Longword displacement deferred	4 bytes

Providing the PC update value eliminates an extra microinstruction in the flow because all PC updates for the opcode must be handled separately by the microcode

In compatibility mode, the hardware determines the addressing mode and whether or not an address calculation is required If the data following the current instruction is required for execution, the number 2 is added to the PC The PC update number is 0 for the following conditions

- A fault is detected (parity error, TB miss, or stall)
- The instruction consists of a single byte opcode
- The decision-point entry is to an execution flow
- The first part done (FPD) flag is set

3.3.1.4 **Memory Data (MD) Byte Shifter** - Figure 3-13 shows a basic example of I-stream data alignment that takes place on a fetch

Instructions are stored in memory, starting in even or odd byte addresses However, all memory references take place on aligned longword boundaries and the memory data (MD) byte shifter receives a longword of data from the MD bus The MD byte shifter right-rotates the longword to align the opcode with byte 0 of the longword unless it already occupies that position in the longword received from memory Instruction buffer address bits IBA (01 00) initially reflect the states of the low-order virtual instruction buffer address bits (VIBA (01:00)) from the CPU data paths that select a longword byte address. IBA (01:00) then determine MD byte shifter rotation as shown in Table 3-14.

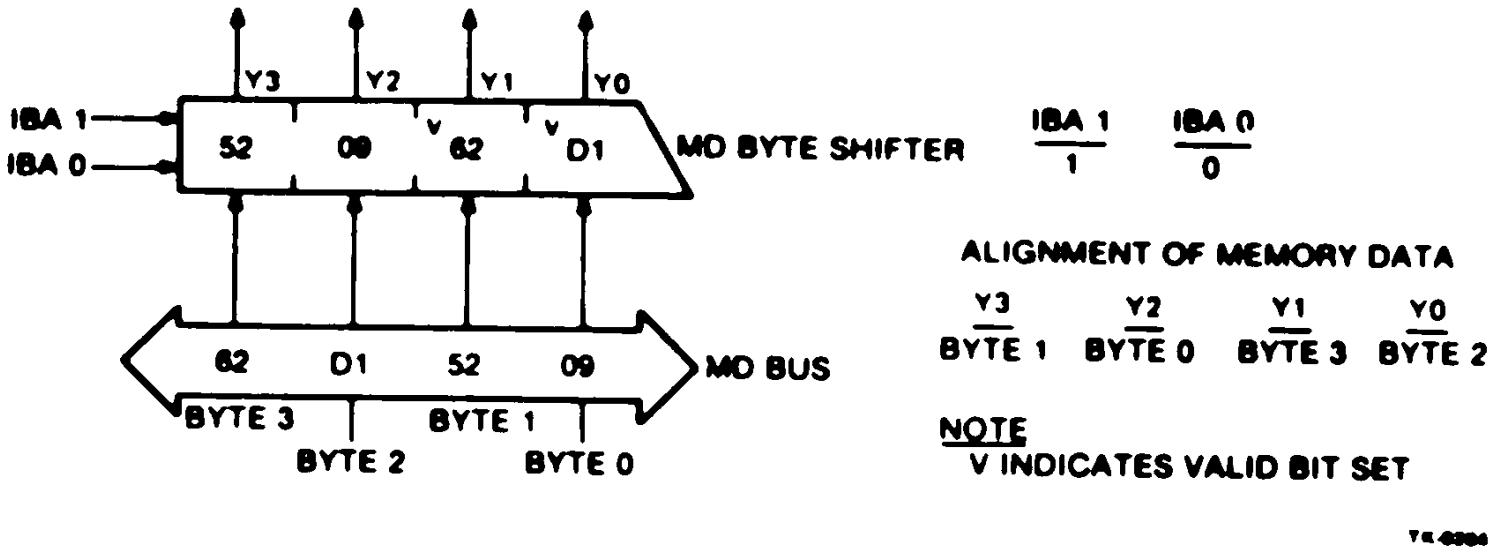


Figure 3-13 Example of a Memory Data Byte Shift

Table 3-14 IBA Control of the MD Byte Shifter

IBA Bits		MD Byte Shifter Outputs			
⟨1⟩	⟨0⟩	Y3	Y2	Y1	Y0
0	0	Byte 3	Byte 2	Byte 1	Byte 0
0	1	Byte 0	Byte 3	Byte 2	Byte 1
1	0	Byte 1	Byte 0	Byte 3	Byte 2
1	1	Byte 2	Byte 1	Byte 0	Byte 3

The function of the MD byte shifter can also be shown by using Figure 3-13 with the following example. With the I-stream data stored in the memory byte locations shown here, assume that the program branches to the location "CHECK".

Memory Address (Byte Locations)	I-Stream Data (Hex Code)	Assembler Notation
200	-	
201	-	
202	-	
203	C0	ADDL #09,R2
204	09	
205	52	
206	D1	CHECK: CMPL (R2),R4
207	62	
208	54	

The program branch results with the instruction buffer address equal to 206. Because the memory reference is on a longword boundary, the lower bits IBA ⟨01:00⟩ of the instruction buffer address are ignored by memory. The fetch from memory results in the MD byte shifter being loaded with four bytes of data beginning at the longword boundary address 204.

The MD byte shifter asserts the data so that byte 0 of the buffer register is loaded with the contents of byte location 206 and byte 1 is loaded with the contents of bye location 207. Bytes 2 and 3 are also loaded with the contents of byte locations 204 and 205 However, only the valid bits for bytes 0 and 1 are set

3.3.1.5 IB Shift Multiplexer (SHF MUX) - The instruction buffer (IB) output shift multiplexers (SHF MUX, shown in Figure 3-9) are controlled by the UIBC field (BUS CS (95:92)) of the CPU microword

As each byte of I-stream data is used, its validity bit is cleared and the byte is considered empty of valid data The contents of the upper register bytes (including their validity bits) are then shifted into the lower byte positions through the input multiplexer (IMUX) inputs, as determined by the instruction mode, opcode, and operand specifier

The shift multiplexer right-shifts bytes by 0, 1, 2, or 4 positions on each microcycle However, a shifted byte is clocked to the buffer register from the input multiplexer only if its valid bit is set (asserted from the shift multiplexer) If the valid bit of a byte is not set, the input multiplexer selects data from the MD byte shifter

The UIBC (03:00) field of the CPU microword controls MD byte shifter rotation as shown in Table 3-15

3.3.1.6 Loading the Instruction Buffer - The following example shows how an instruction (ADDL #09,R2) is loaded into the instruction buffer from memory, using the MD byte shifter and shift multiplexer The example program in the following two formats is shown with both byte address and longword address references

Byte Memory Addresses	Byte Contents	Assembler Notation
200	D0	MOVL (R3),(R4)+
201	63	
202	84	
203	C0	ADDL #09,R2
204	09	
205	52	
206	D1	CMPL (R2),R4
207	62	
208	54	
209	13	BEQL,DONE
20A	05	
20B	01	NOP
20C	90	MOVB (R7),R8
20D	67	
20E	58	
20F	B4	CLRW R4

Longword Memory Address	Longword Contents
200	C0 84 63 D0
204	62 D1 52 09
208	01 05 13 54
20C	B4 58 67 90

Table 3-15 UIBC Field Control of the MD Byte Shifter

UIBC Value	Function	Comment
0	NOP	Shifts by 0. Does not affect shift multiplexer or data input to IMUX.
1	STOP	Prevents further memory requests by the instruction buffer.
2	FLUSH	Clears all buffer register valid bits.(Used for conditions that cause a change in the PC jump or branch). Also loads the data path VIBA register.
3	START	Enables prefetch operations by the instruction buffer.
4	CLR 0,1	In compatibility mode, the next instruction is shifted over the current instruction in bytes 0 and 1. This function is also performed in native mode to optimize short literal to register and register to register transfers and to execute 16-bit branch destinations.
5	CLR 2,3	In compatibility mode, the 16-bit displacement or literal data in bytes 2 and 3 is shifted into bytes 0 and 1.
6	Reserved	
7	READ BDEST	Used during the ACB, AOB, SOB, BBS, and BBC instructions to indicate branch displacement specifiers.
8-B	Reserved	
C	CLR 0	In native mode, the current opcode in byte 0 is shifted over with the next opcode.
D	CLR 1	In native mode, the operand specifier in byte 1 is shifted over with new data. This function is performed for instructions using displacement mode addressing, absolute addressing or long literals. In these modes, the operand specifier is the last byte removed from the register during operand evaluation.
E	CLR 0,1, 2,3	This function is performed in compatibility mode to optimize instructions that execute literal data to register transfers.
F	CLR 1-5 Conditional	This function is performed in native mode mode to remove long literals, displacement data, or specifiers from the buffer register. The op code and/or specifier evaluation indicates if the data is 8, 16, or 32 bits. In addressing modes that do not have I-stream data other than the specifier, the specifier itself is cleared from the buffer register

3.3.1.6.1 First Memory Fetch – The instruction ADDL #09,R2 does not begin on a longword boundary and IBA ⟨01:00⟩ equal 3. However, virtual address bits ⟨01:00⟩ are ignored on a memory reference. Data is retrieved on a longword boundary and the first memory fetch loads four bytes of data into the MD byte shifter from memory address 200. The MD byte shifter then aligns the data so that the opcode (C0) is loaded into byte 0 of the buffer register and the valid bit is set as shown in Figure 3-14. All four bytes are loaded but the valid bit, set only in byte 0, prevents it from being written on the next memory fetch.

After the first fetch, the instruction buffer address is incremented by four and equals 207 but the two lowest address bits remain the same. (Adding four to the VIBA register increments the address by a longword and does not affect the byte position.)

3.3.1.6.2 Second Memory Fetch – The second memory fetch then loads four bytes of data beginning at longword boundary 204. With bits IBA ⟨01:00⟩ at the same value, the MD byte shifter aligns the data as it did in the first fetch and loads it, setting the validity bits as shown in Figure 3-15.

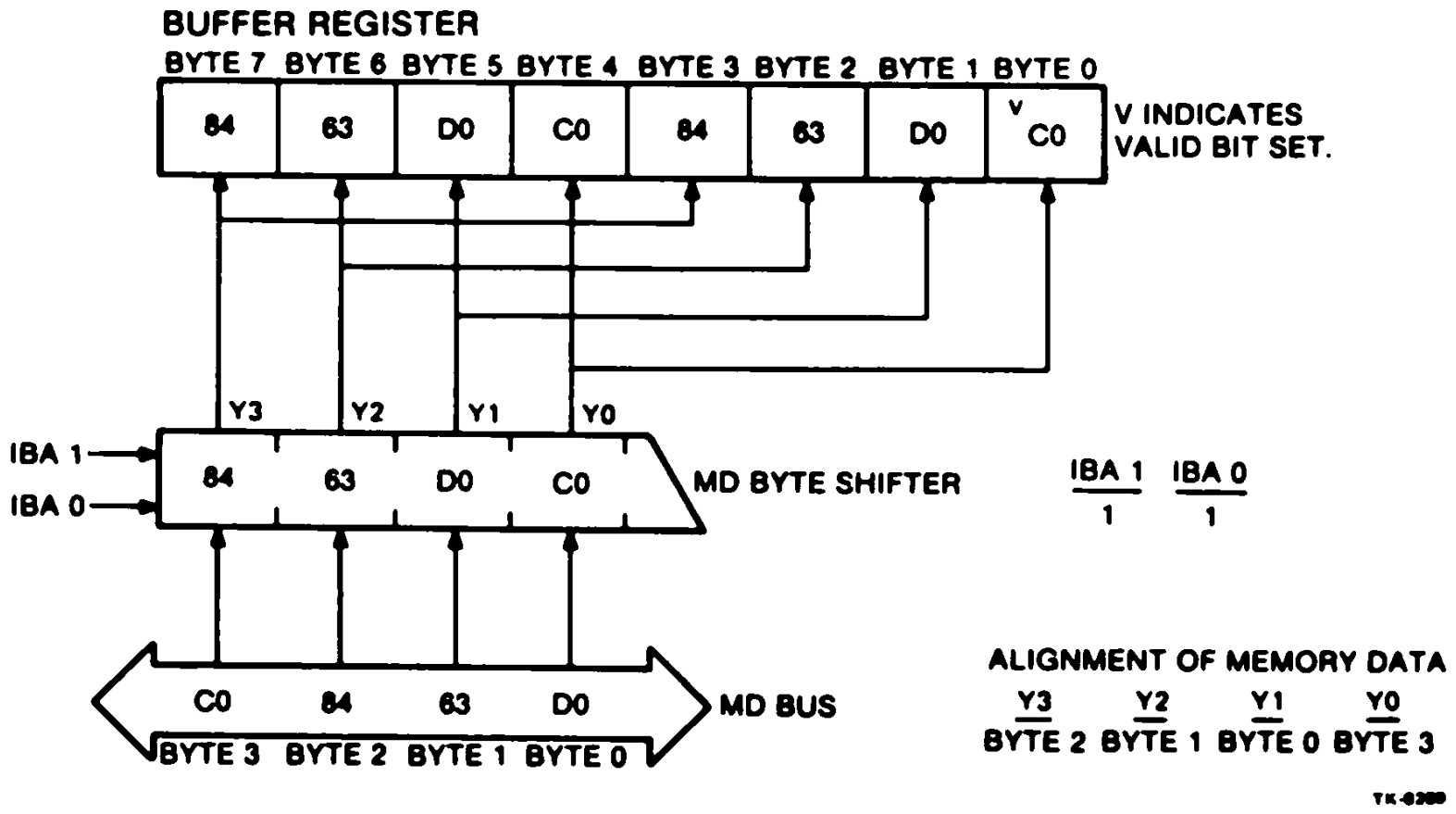


Figure 3-14 Result of the First Memory Fetch

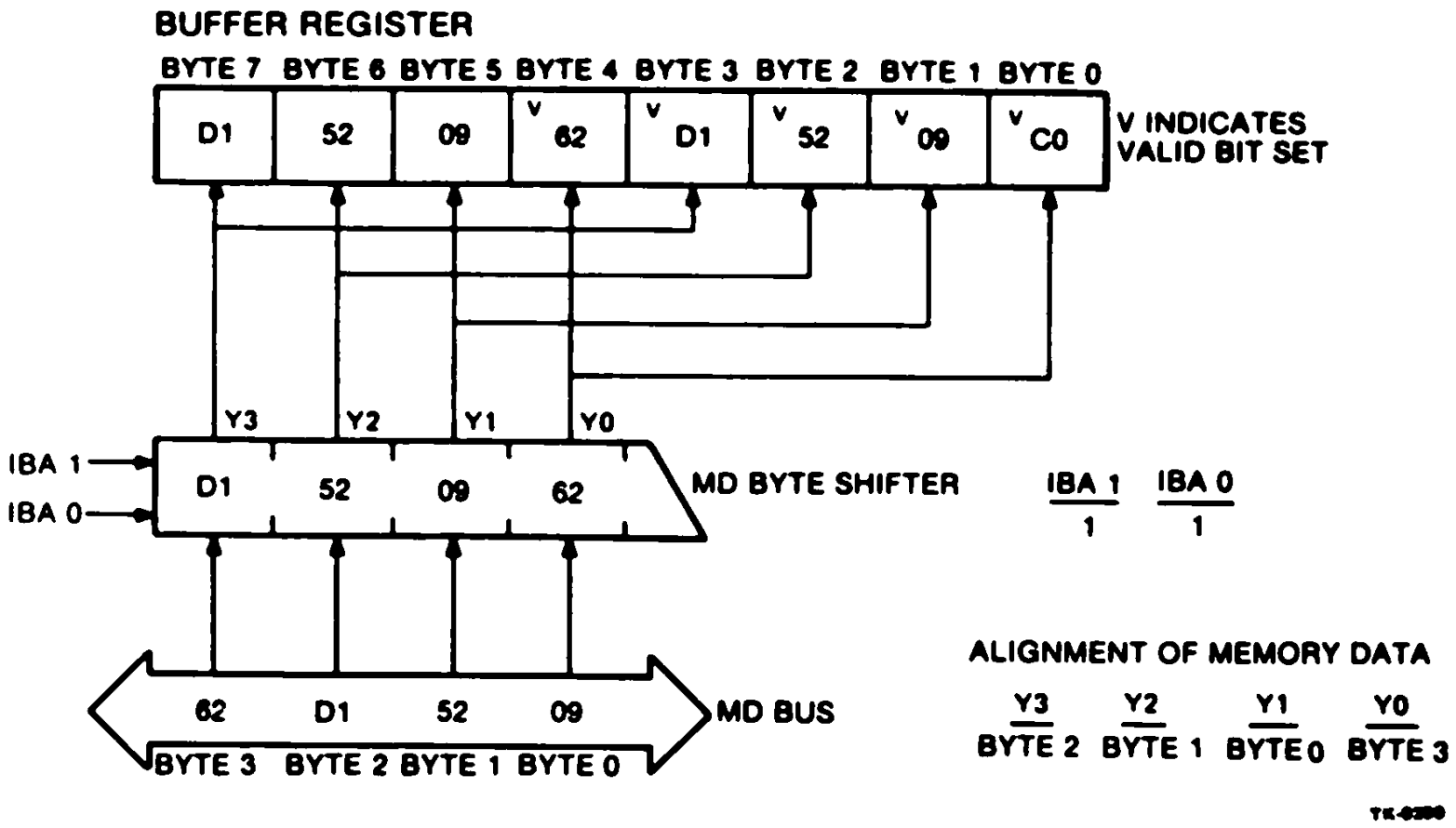

BUFFER REGISTER
BYTE 7 BYTE 6 BYTE 5 BYTE 4 BYTE 3 BYTE 2 BYTE 1 BYTE 0
D1 52 09 V 62 V D1 V 52 V 09 V C0
V INDICATES VALID BIT SET
Y3 Y2 Y1 Y0
IBA 1
IBA 0
D1 52 09 62
MD BYTE SHIFTER
IBA 1 IBA 0
1 1
ALIGNMENT OF MEMORY DATA
Y3 Y2 Y1 Y0
BYTE 2 BYTE 1 BYTE 0 BYTE 3
62 D1 52 09
MD BUS
BYTE 3 BYTE 2 BYTE 1 BYTE 0

Figure 3-15 Result of the Second Memory Fetch

Byte 0 was not written its valid bit is set as a result of the first fetch. The byte 0 input multiplexer selects data from the shift multiplexer rather than from the MD byte shifter and the data is the same as the current contents of byte 0 because of a shift by zero (UIBC field equals 0) was selected by the microword. (The number of valid bytes loaded from the first fetch depends on the instruction buffer byte address. The second fetch always loads four bytes of valid data.)

After the second fetch, the instruction buffer has all the necessary data to execute the instruction ADDL #09,R2. At instruction decode time (microcode IRD state), the literal data (09) in register byte 1 is selected by the data multiplexer (DMX) and transferred to the Q register of the data path on the ID bus. This transfer leaves byte 1 vacant, clearing the valid bit. At the end of the IRD state, a shift by 1 causes all valid data to be shifted by one position in the buffer register as shown in Figure 3-16. Byte 0 (the opcode) is not affected because its valid bit is still set and all validity bit states are shifted with the data from the higher byte locations. (Bytes 4 through 7 contain invalid data.)

The low address bits IBA ⟨1:0⟩ are incremented by one because the data in the buffer register has moved by one byte position. These address bits keep track of the end of the buffer and are incremented independently of address bits ⟨31:02⟩. The instruction buffer longword address bits ⟨31:02⟩ are incremented by 4 after the last memory fetch and result with the address equal to 208.

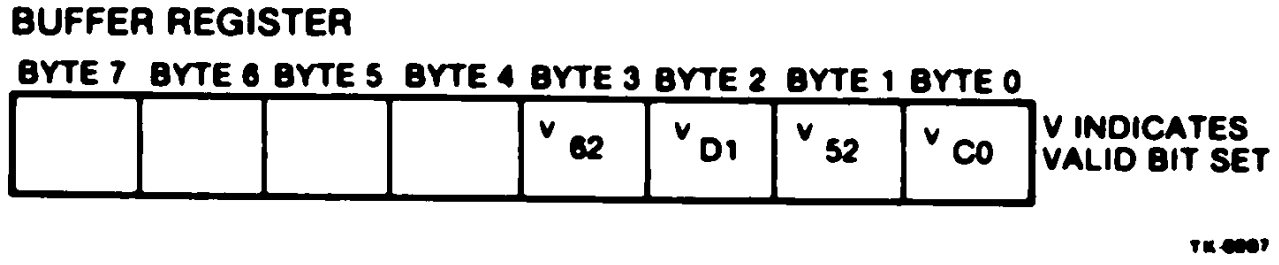


Figure 3-16 Result of a 1-Byte Right Shift

3.3.1.6.3 Third Memory Fetch - The next memory fetch begins on a longword boundary. As shown in Figure 3-17, this data is also loaded without being rotated because IBA (01:00) are equal to zero.

With the valid bit set in byte 7, the instruction buffer is full and the instruction buffer address is not incremented by 4. It will not be incremented until data is shifted out of the buffer and the valid bit in byte 7 is cleared.

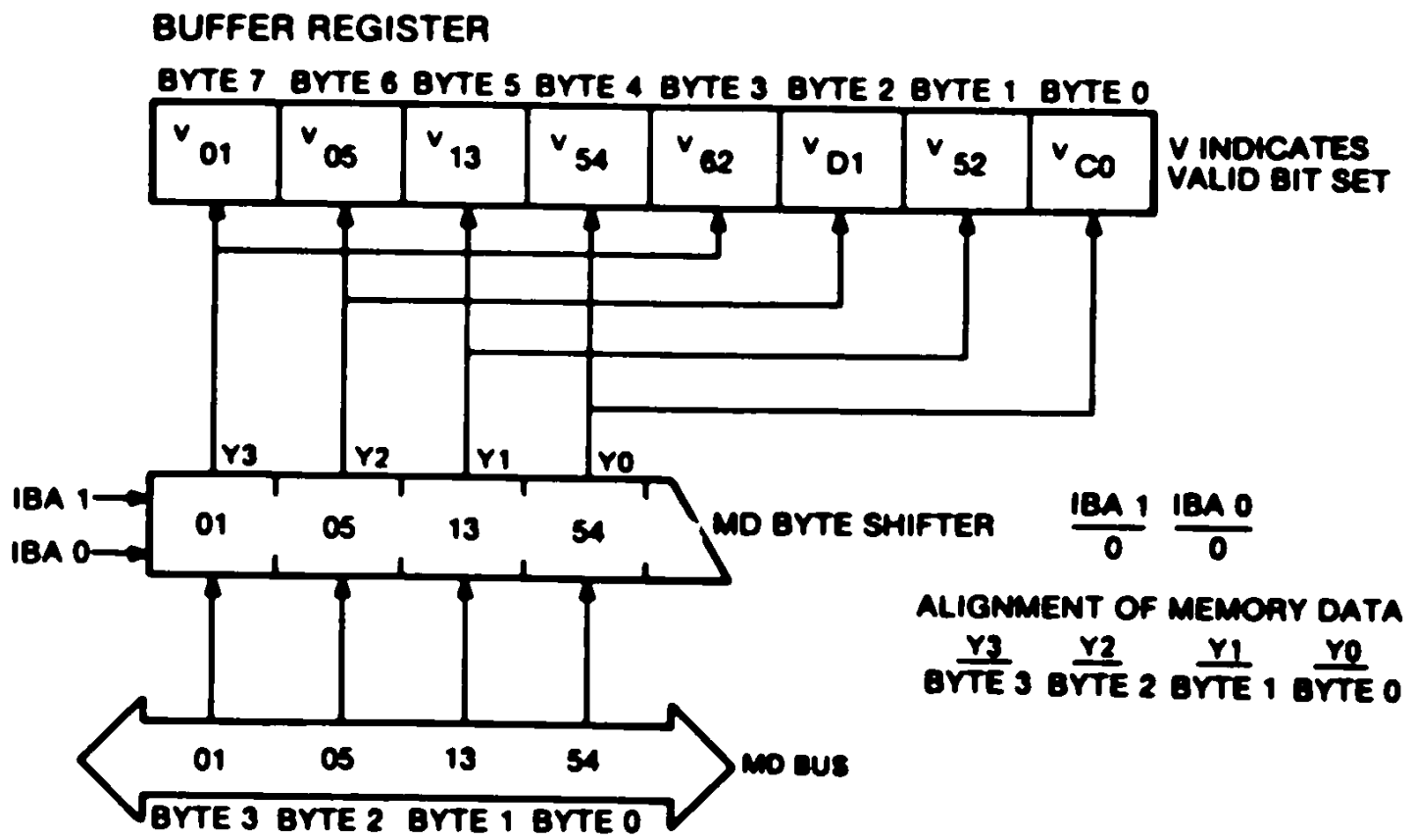


Figure 3-17 Result of the Third Memory Fetch

At the next fork entry in the microcode, byte 0 (opcode) and byte 1 (operand specifier) are removed (their valid bits are cleared). The microcode specifies a two-byte shift and the remaining valid data is shifted and stored as shown in Figure 3-18.

Bytes 0 through 5 receive the data from bytes 2 through 7 from the shift multiplexers and the valid bits in byte 6 and 7 are cleared. The instruction buffer longword address (IBA (31:02)) is now incremented by 4 because the buffer register successfully fetched data during the previous cycle but the valid bit in byte 7 was not clear. A value of 2 is also added to IBA (1:0) because two bytes of data were cleared from the buffer register.

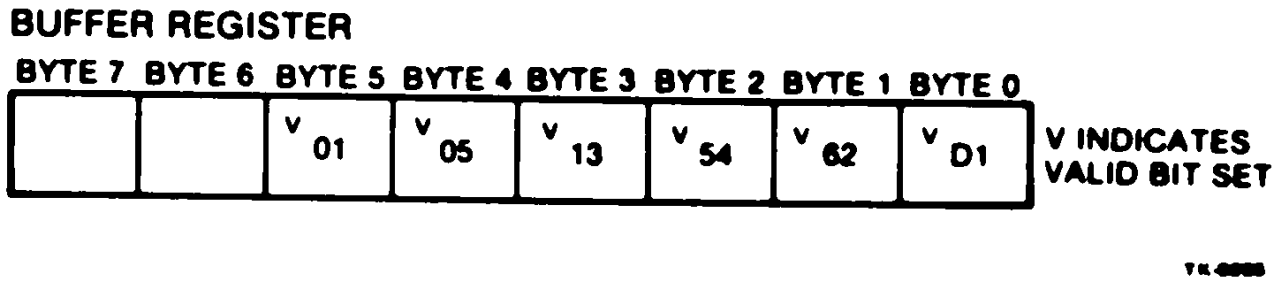


Figure 3-18 Result of a 2-Byte Right Shift

3.3.1.6.4 Fourth Memory Fetch – The next fetch loads data from longword address 20C. This data is rotated as shown in Figure 3-19 because the two low address bits are now equal to 2. In the instruction CMPL (R2), R4 could have been executed without the fourth memory fetch, however, because the instruction buffer contained all the required information when the data was last shifted by two bytes as shown in Figure 3-18.

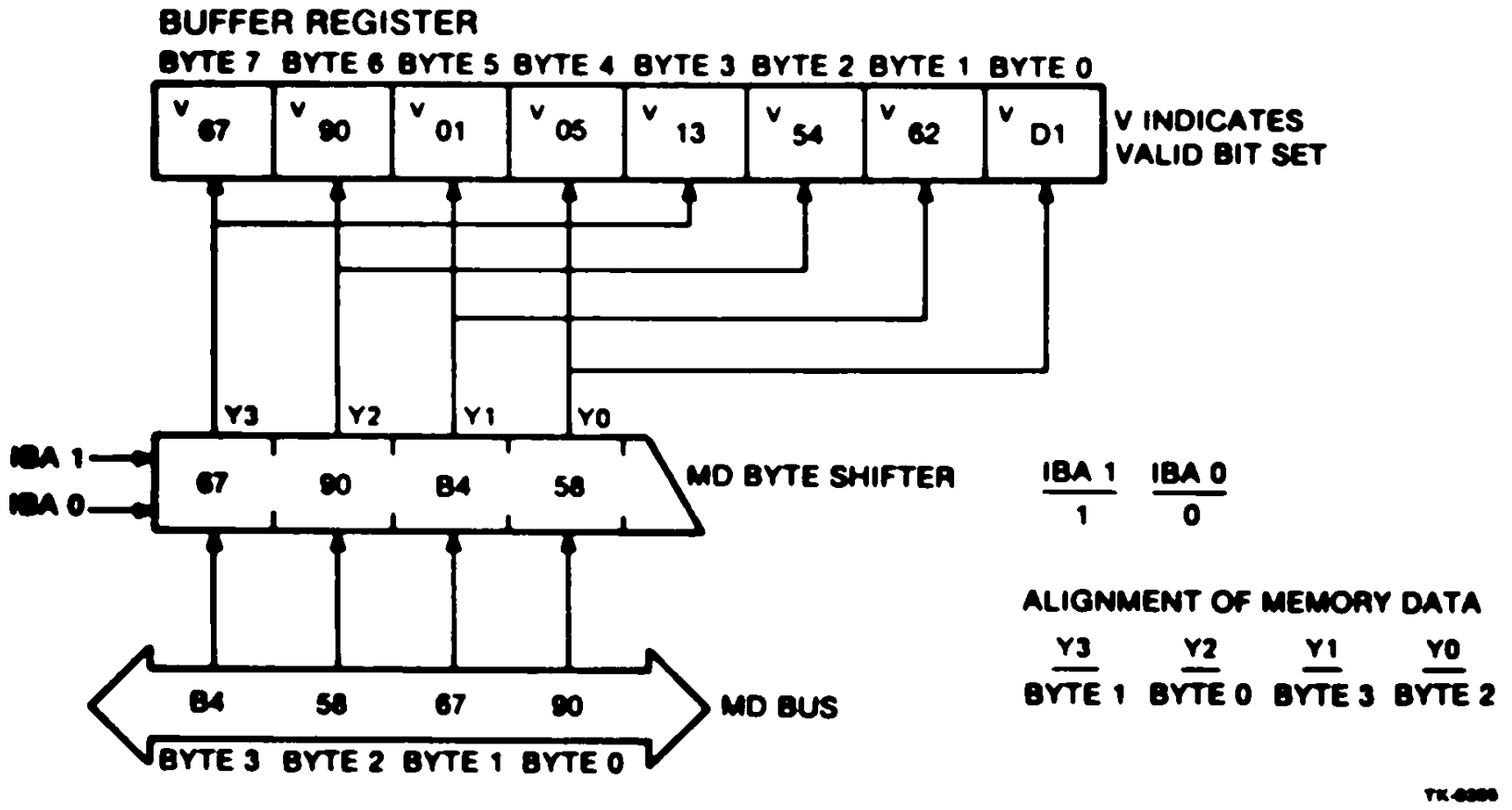


Figure 3-19 Result of the Fourth Memory Fetch

3.3.1.7 Data Multiplexer (DMX) – The data multiplexer (DMX) shown in Figure 3-20 decreases the instruction execution time for certain addressing modes in both VAX-11 and PDP-11 modes. When the opcode and operand specifier have been decoded, the DMX is then used to sort further I-stream data from the buffer register and arrange it in a more usable form.

The DMX transfers the buffer register contents to other areas of the CPU on the internal data (ID) bus, sign-extending or shifting the data if necessary. This optimizes the evaluation of displacement and literal specifiers by allowing the direct transfer of information from the buffer register.

Buffer register data asserted to the ID bus is written to the data path Q register from where it is written to the specified destination. The DMX is selected as the source for ID bus data if the address lines (ID RIGHT ADDR (05:00)) specify the DMX (the hex address equals 00) and the control line ID RIGHT WRITE is not asserted.

Selection of data by the DMX requires that the op code and specifier of the instruction first be decoded. Buffer register byte data is then gated to the ID bus according to the addressing mode as shown in Table 3-16. If the operand is a floating data type for the short literal mode, the short literal format is as shown in Figure 3-21 and the DMX formats the data as shown in Figure 3-22.

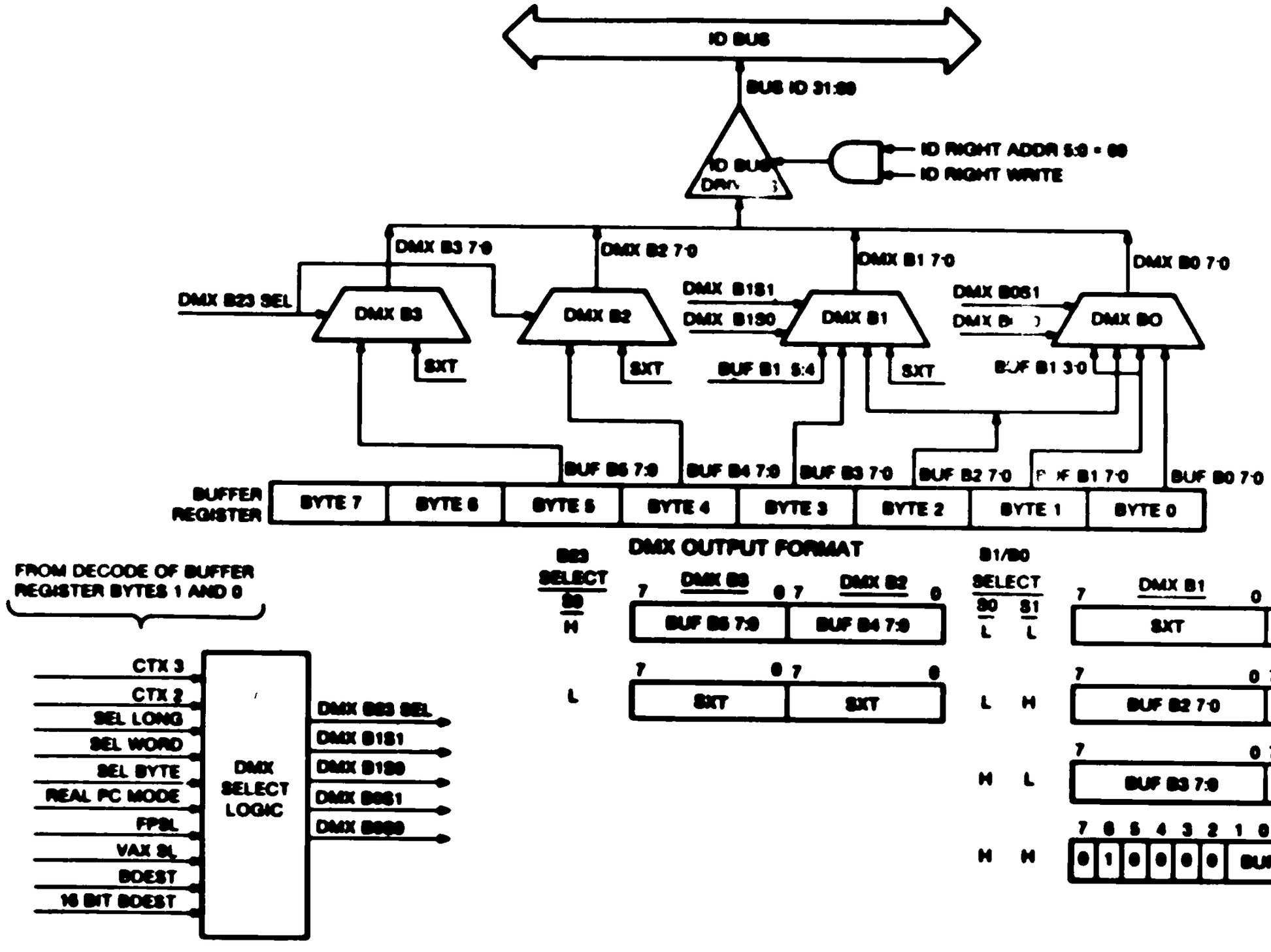


Figure 3-20 Data Multiplexer (DMX) Logic

Table 3-16 Address Mode Control of DMX Data

Addressing Modes	ID Bus Data	Comments
General Addressing Modes		
Short Literal	B1 = (07:00)	Zero Extended
Indexed	N/A	
Register	N/A	
Register deferred	N/A	
Autoincrement (Rn = PC)	(B5:B2) = (31:00)	1, 2, or 4 bytes
Autoincrement deferred (Rn = PC)	(B5:B2) = (31:00)	32 bit address
Byte displacement	B2 = (07:00)	Sign-extended on bit (07)
Byte displacement deferred	B2 = (07:00)	Sign-extended on bit (07)
Word displacement	(B3.B2) = (15:00)	Sign-extended on bit (15)
Word displacement deferred	(B3 B2) = (15:00)	Sign-extended on bit (15)
Longword displacement	(B5 B2) = (31:00)	
Longword displacement deferred	(B5 B2) = (31:00)	
Branch Addressing Modes		
8-bit byte displacement	B1 = (07:00)	Sign-extended on bit (07)
16-bit word displacement	(B2 B1) = (15:00)	Sign-extended on bit (15)

In compatibility mode, the DMX selects data from the buffer register for the following types of PDP-11 instructions:

- Branch instructions, stored in the buffer register, as shown in Figure 3-23. The DMX selects the branch displacement from byte 0, left shifts the data by one bit and sign extends the data on bit (07).

- Instructions followed by 16-bit displacement or 16-bit literal data, stored in the buffer register in the format shown in Figure 3-24. The DMX selects the data from bytes 2 and 3 of the registers to be transferred onto the ID bus. The DMX shifts these bytes so that they are transferred as ID bits (15:00), sign-extended on bit (15).

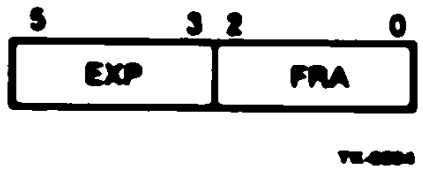


Figure 3-21 Floating Point Short Literal Format

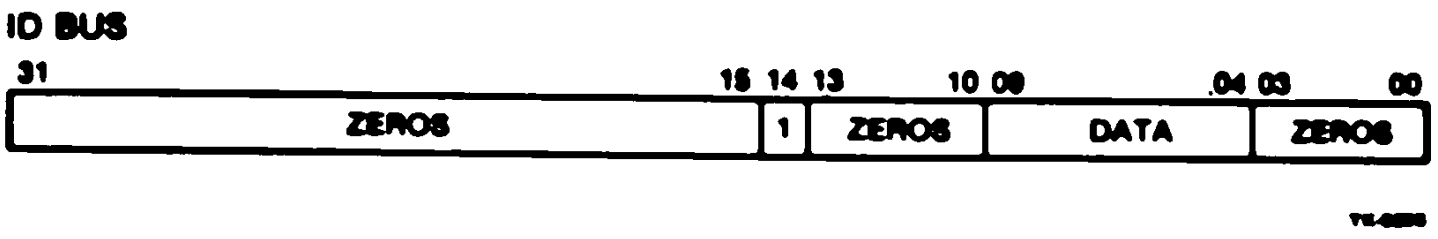


Figure 3-22 DMX Format of a Floating Point Short Literal

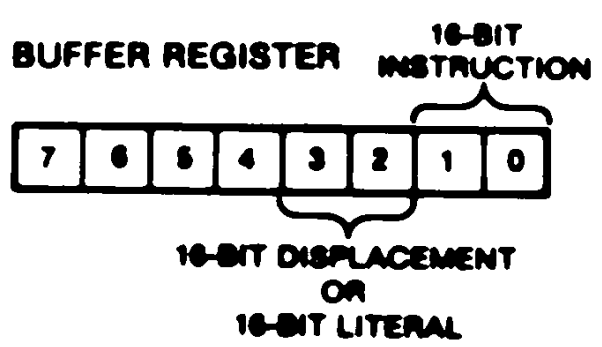


Figure 3-23 PDP-11 Branch Instruction Format in the Buffer Register

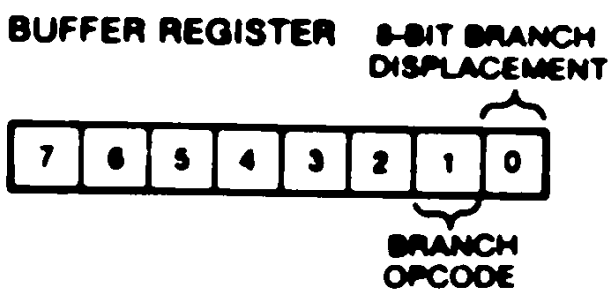


Figure 3-24 PDP-11 Instruction Format in the Buffer Register

3.3.2 Instruction Decoder (IRC) Module

The IRC module controls the decision point branches of the CPU microprogram. It contains combinational gating and ROMs shown in Figure 3-25 that interpret the instruction stream (I-stream) data from bytes 0 and 1 of the instruction buffer register (IDP module).

When microprogram flow reaches a decision point, it specifies a decision point branch (a value of 3) in the subroutine (USUB) field of the microword. While the microsequencer (USC) module asserts BUS UPC (12:08) from the UJMP field of the microword, it disables its address drivers for BUS UPC (07:00) and enables the UPC ADRS MUX outputs in the IRC instruction decode logic. The UPC ADDRS MUX then enables BUS UPC (07:00) for the next microaddress (NUA), asserting the entry point for the current instruction or operand specifier. If an error or service condition is present, the UPC ADRS MUX alters the entry point to select the service routine that handles the condition.

At instruction register decode (IRD) time, the I-stream data in bytes 0 and 1 of the buffer register are decoded by the VAX execution ROM in native mode or by the PDP-11 execution ROM in compatibility mode. Compatibility mode decoding directs the microprogram to a flow that either executes the instruction or evaluates the source or destination mode of the operand. Native mode decoding directs the microprogram to execute a one-byte instruction, selects the branch decode logic for a branch condition instruction or, on variable length instructions, selects the specifier decode logic for operand specifier evaluation.

3.3.2.1 Execution Point Counter – A three-bit execution point counter, shown in Figure 3-26, keeps track of the decision points that have been entered during the execution of each instruction. The counter selects ranges of VAX or PDP-11 execute ROM address space while bytes 0 and 1 of the buffer register select a specific execute ROM address.

The counter is set to zero when the opcode is removed from byte 0 of the instruction buffer (or by a flush of the instruction buffer) indicating that the current instruction has been executed and the next instruction can start. The counter is incremented at the start of each CPU cycle (CPT 0) if one of the following conditions is met:

- The operand specifier has been cleared from the buffer register and was not in index mode (I mode).
- The microword USUB field selects the next microaddress on the instruction decode (IB ADVANCE) and a STALL is not in effect.

The first part done (FPD) signal sets the counter to 7 (execution point 7). When the FPD flag is set by the microword (the UMSC field equal to 9), it indicates that an interrupt was received during the execution of an interruptable instruction. When the interrupt service routine completes, the instruction is fetched again. Instead of evaluating the operand specifiers again, however, the microprogram re-enters execution flow. The FPD flag is then cleared (the UMSC field equal to 8), allowing evaluation of the next instruction.

3.3.2.2 VAX Execution ROMs and Control Word – The VAX execution ROMs generate the 12-bit VAX control word shown in Figure 3-27. The VAX control word asserts two groups of bit fields, CONTEXT (03:00) and VAX DECODE (07:00), as described in the following paragraphs.

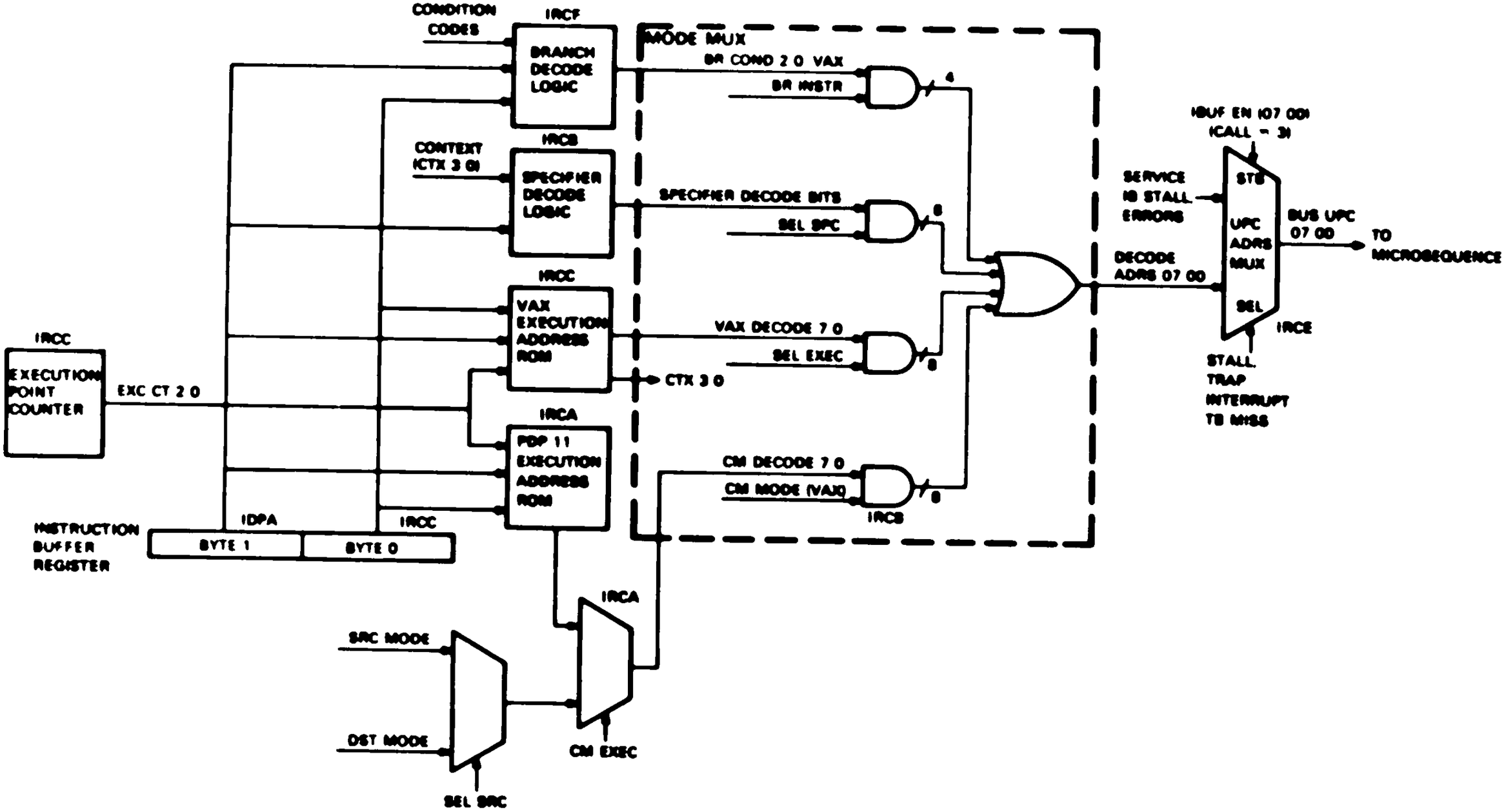


Figure 3 25 Instruction Decode Logic

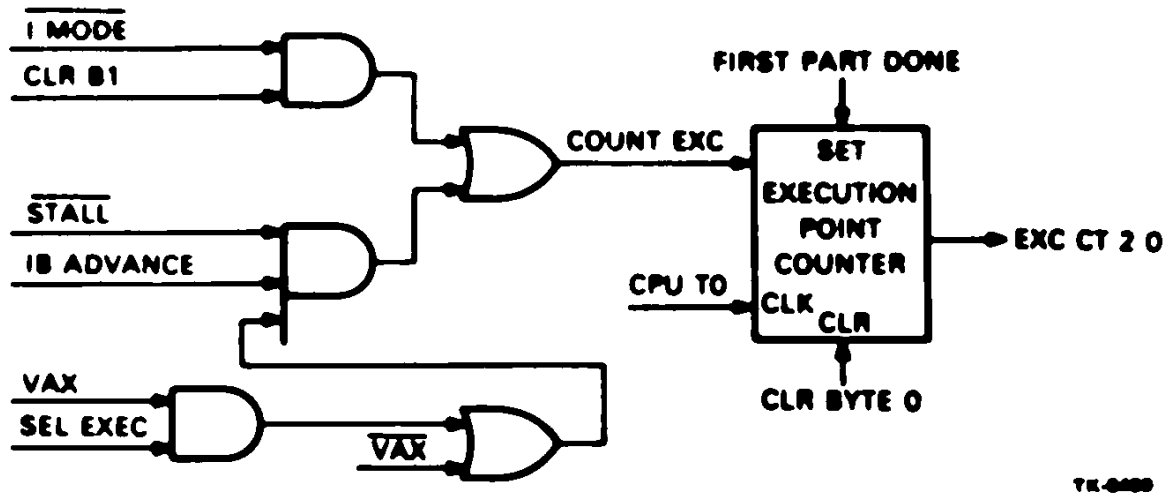


Figure 3-26 Execution Point Counter

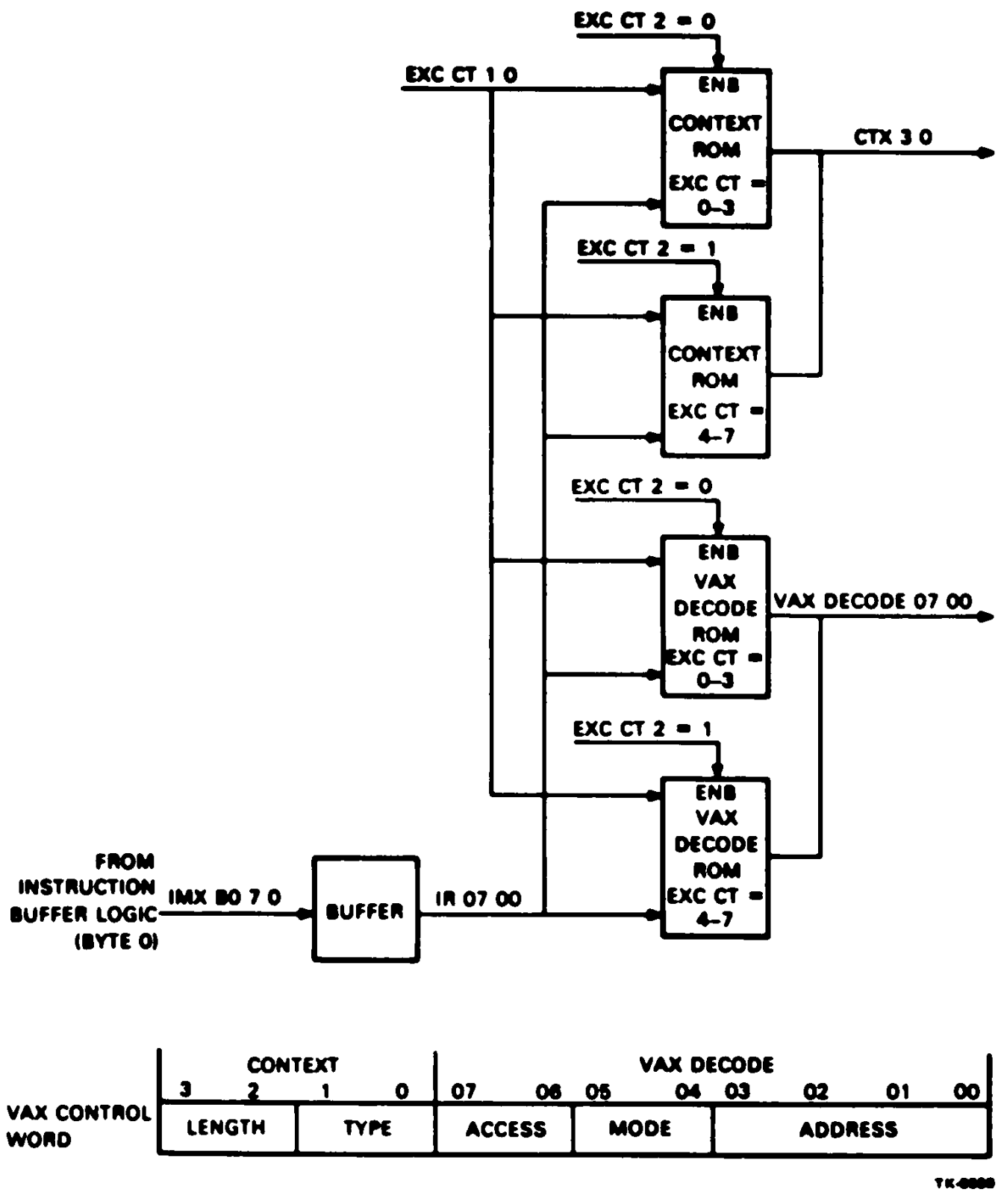


Figure 3-27 VAX Execute ROMs and Control Word Format

VAX DECODE Group

- *Mode Field* - Bits (05:04) control the mode multiplexer as shown in Table 3-17.

- *Address Field* - Bits (03:00) are used only when the mode field specifies an execute if R mode (1) or select execute (3) function. The field is then used as part of the one's complement of the VAX DECODE field (bits (07:00)) which is asserted as the execution microaddress.

- *Access Field* - As shown in Table 3-18, bits (07:06)), either select the branch decode logic or define the type of transfer (read, write, or modify) specified by the instruction at an execution point If the mode field is equal to 1 or 3, however, the access field is used as part of the one's complement of the VAX DECODE field (bits (07:00)) that is asserted as the execution microaddress.

Table 3-17 Mode Field Control of the Mode Multiplexer

Mode Bits (05)	Mode Bits (04)	Mode Field Function
0	0	*Select Specifier* enables the specifier decode logic. The addressing mode determines the microaddress bits to be generated as described in Section 3.3.2.4.
0	1	*Execute if R Mode* makes the following decisions from the operand specifier in byte 1 of the IDP instruction buffer register. - Byte 1 specifies register mode; the one's complement of the VAX DECODE field (bits (07:00)) are asserted as the execution microaddress. - Byte 1 does not specify register mode, the specifier decode logic is selected, generating the execution microaddress as described in Section 3.3.2.4.
1	0	*Optimize* enables the specifier decode logic and modifies the specifier address for a short literal to register transfer or a register to register transfer as described in Section 3.3 2.5 If optimimum conditions are not met, however, the specifier microaddress is not modified
1	1	*Select Execute* asserts the one's complement of the VAX DECODE field (bits (07:00)) as the execution microaddress.

Table 3-18 Access Field Definitions

Access Bits (07)	(06)	Access Field Function
0	0	*Branch* – Enables the branch decode logic only at execution point zero. At any other execution point, this code is part of the execution address.
0	1	*Read* – Indicates a transfer from cache memory to the D register in the CPU data paths.
1	0	*Write* – Indicates a transfer from the D register in the CPU data paths to cache memory.
1	1	*Modify* – Indicates a read-modify-write, informing the translation buffer of a read with a write check.

CONTEXT Group

- *Operand Length and Type Fields* – The context ROMs decode the operand length and type as shown in Table 3-19 and supply the information to the specifier decode logic. The operand type specifications are defined in Table 3-20.

Table 3-19 Operand Length and Type Bit Field Definitions

Length Bits (03)	(02)	Operand Length	Type Bits (01)	(00)	Operand Type
0	0	Byte	0	0	Integer
0	1	Word	0	1	Floating
1	0	Long	1	0	VSRC*
1	1	Quad	1	1	ASRC**

* This code is used only in field instructions that use the calculated address for the operand. Register mode may be specified in address access type operand specifiers.

** This code is used in all other instructions that use the calculated address for the operand. Register mode may not be specified in address access type operand specifiers.

Table 3-20 Operand Length and Type Definitions

Operand Length	Type	Data Type
Byte	Integer	Byte
Word	Integer	Word
Long	Integer	Longword
Quad	Integer	Quadword
Byte	Floating	Undefined
Word	Floating	Undefined
Long	Floating	Floating (F_floating)
Quad	Floating	Double floating (D_floating)

3.3.2.3 Mode Multiplexer (Mode Mux) - The mode multiplexer shown in Figure 3-28 selects the source for the output address lines DECODE ADRS ⟨07:00⟩ for assertion to the UPC ADRS MUX. The address source selected is determined by the operating (native or compatibility) mode, the instruction opcode, and the state of the execution point counter. In either native or compatibility mode, the BRANCH INSTR line is only asserted at execution point zero.

In native mode, the VAX control word (VAX DECODE ⟨07:00⟩) determines the assertion of one of the mode multiplexer enable lines shown in Table 3-21.

Table 3-21 Native Mode Control of the Mode Multiplexer

Enable Line	Description
SEL SPC	*Select Specifier* - Asserted for operand specifier evaluation or for double operand optimizing as described in Section 3.3.2.5.
SEL EXEC	*Select Execute* - Asserted when the necessary operand specifier evaluations have taken place and the instruction can be executed. The one's complement of VAX DECODE ⟨07:00⟩ are asserted as the execute microaddress.
BR INSTR	*Branch Instruction* - Inclusive-ORs the branch condition bits with the one's complement of VAX DECODE ⟨03:00⟩ (address field). This forms the four low-order bits of the execute microaddress while the four high-order bits assert the one's complement of VAX DECODE ⟨07:04⟩.

In compatibility mode, the PDP-11 execute ROM outputs (CM DECODE ⟨07:00⟩) select the next microaddress, as described in Section 3.3.2.11. For a PDP-11 branch instruction, the condition bits are inclusive-ORed with the one's complement of CM DECODE ⟨03:00⟩. This forms the four low-order bits of the execute microaddress while the four high-order bits assert the one's complement of CM DECODE ⟨07:04⟩.

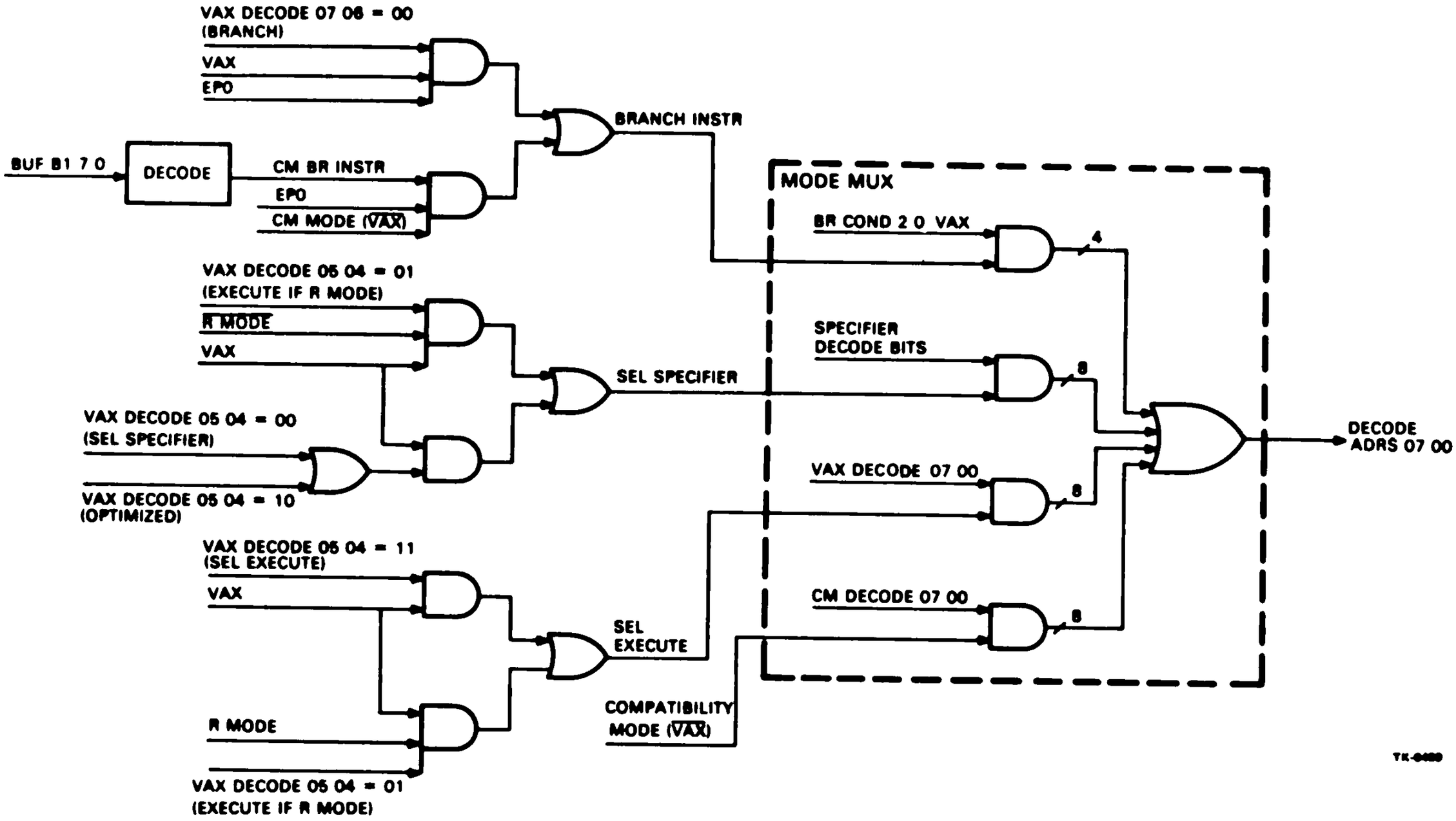
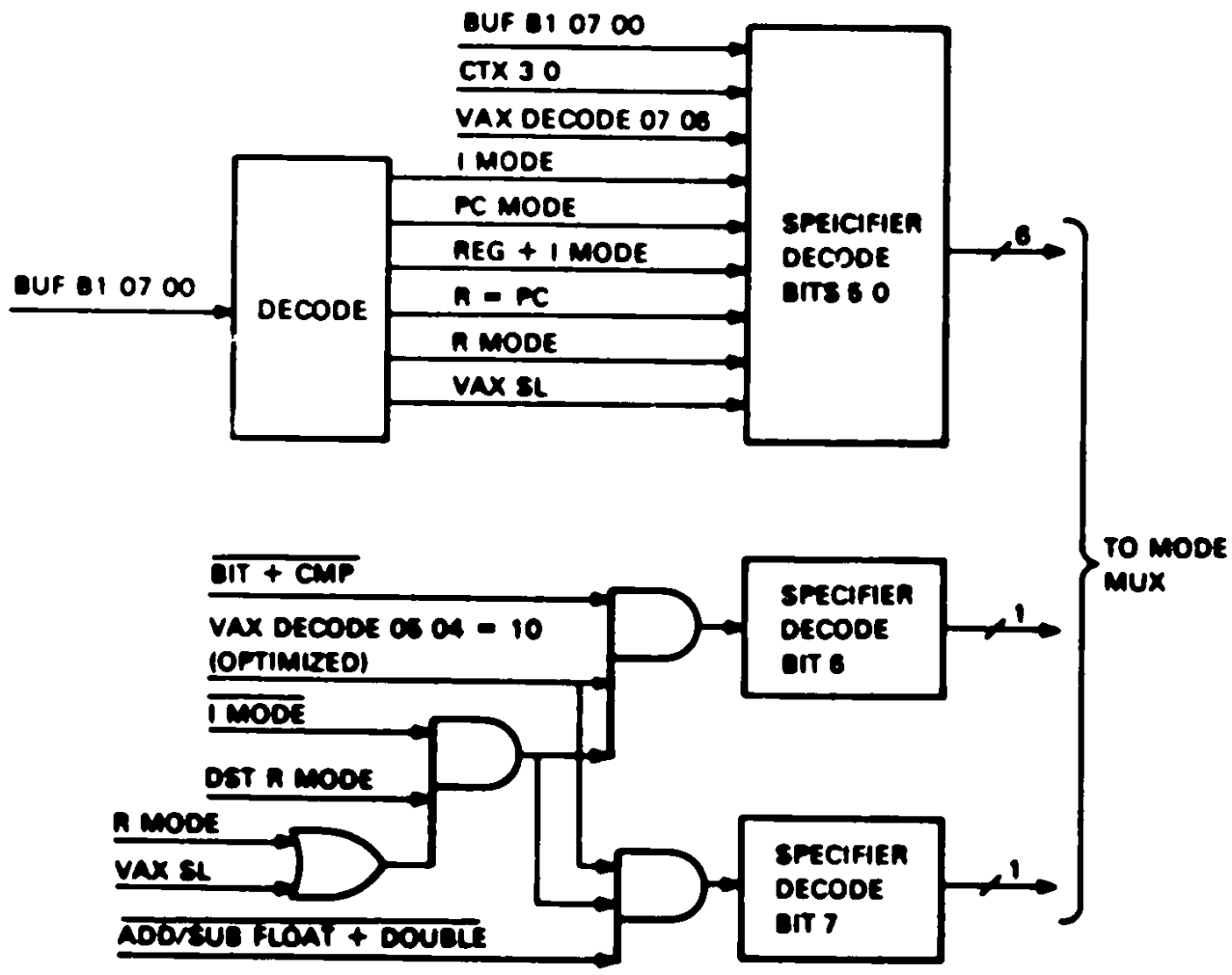


Figure 3-28 Mode Multiplexer Selection

3.3.2.4 Specifier Decoding - The specifier decode logic shown in Figure 3-29 is selected for operand specifier evaluation or for double operand optimizing. At each execution point, the mode field of the VAX control word determines whether the specifier decode logic is selected. Bits (07:60) are normally zero unless the optimization conditions are met. Bits (05:00) are asserted as a function of the addressing mode, the context field of the VAX control word, and the operand's use of the PC. The addresses generated by bits (05:00) are shown in Table 3-22. The abort addresses are caused by the conditions shown in Table 3-23.



OPTIMIZATIONS

SPECIFIER DECODE

BIT 7	BIT 6	COMMENT
0	0	OPTIMIZED CONDITIONS NOT MET
0	1	F1 CLASS (ADDX2, SUBX3, ETC.)
1	0	R CLASS (BIT X, CMPX)
1	1	M CLASS (ADDXX, SUBXX, ETC.)

Figure 3-29 Specifier Decode Logic

Table 3-22 Specifier Decode Outputs

Addressing Mode Hex	Assembler	Address Formed by Bits (05:00) R ≠ PC	R = PC	QUAD	ABORT
0	S_#	00	00	02	01/03
1	S_#	00	00	02	01/03
2	S_#	00	00	02	01/03
3	S_#	00	00	02	01/03
4	I	0C	1C	-	1D
5	R	04	14	6/16	7/17, 5/15
6	(R)	08	18	-	-
7	-(R)	0A	1A	-	-
8	(R)+	09	19	1F	-
9	@(R)+	0B	1B	-	-
A	D8	0D	0D	-	-
B	@D8	0F	0F	-	-
C	D16	0D	0D	-	-
D	@D16	0F	0F	-	-
E	D32	0D	0D	-	-
F	@D32	0F	0F	-	-

3.3.2.5 Instruction Optimization - The execution of certain instructions are optimized by eliminating specifier evaluations for certain addressing modes and instruction classes.

For instructions such as ADDW2 and BISW2, double operand optimizations are performed when the first operand specifier is in short literal or register mode and the second operand specifier is decoded as register mode. The signal DST R MODE is asserted to the specifier decode logic to enable bits (07:06) as shown in Figure 3-29.

For the double operand instructions that can be optimized, the mode field of the VAX control word equals 2 (optimize) at execution point zero. If the specifiers are already in an optimized form (a short literal to register or a register to register transfer) the microaddress output is modified by changing the value of specifier decode bits (07:06).

The class of an optimized instruction determines the value of specifier decode bits (07:06). If the optimize conditions are not met, bits (07:06) are zeros and the specifier microaddress is not modified.

If the first specifier in a double operand instruction is not in short literal or register mode, the operand specifier is evaluated at execution point zero. At execution point one, the second specifier is checked to determine if it is in register mode. If it is, the instruction can be executed. If it is not, a second specifier evaluation must be performed before it is executed. Figure 3-30 shows the general flow of the double operand instructions that can be optimized.

Single operand optimizations are implemented for instructions such as INCB and TSTB if the operand specifier is register mode. For the single operand instructions that can be optimized, the mode field of the VAX control word equals 1 (execute if R Mode) at execution point zero. If the operand specifier is in register mode, the one's complement of VAX DECODE bits (07:00) are used to form the microaddress. If the specifier is not in register mode, the specifier decode logic is selected as the address source. Figure 3-31 shows the general flow of single operand instructions that can be optimized.

Table 3-23 Abort Address Conditions

Address	Abort Conditions
01	• Writing to a short literal • I mode followed by a short literal • Using a short literal as VSRC or ASRC
03	• Quad context and - Writing to a short literal - I mode followed by a short literal - Using a short literal as VSRC or ASRC
05	• Using register mode as an ASRC • I mode followed by register mode
07	• Quad context and - Using register mode as an ASRC - I mode follwed by register mode
14	• Register mode and Rn select the PC
15	• Rn selects the PC and - Using register mode as an ASRC - I mode followed by register mode
16	• Rn selects the PC with quad context
17	• Quad context, Rn selects the PC, and. - Using register mode as an ASRC - I mode followed by register mode
18	• Register deferred mode and Rn selects the PC
1A	• Autodecrement mode and Rn selects the PC
1C	• I mode and Rn selects the PC
1D	• I mode followed by I mode

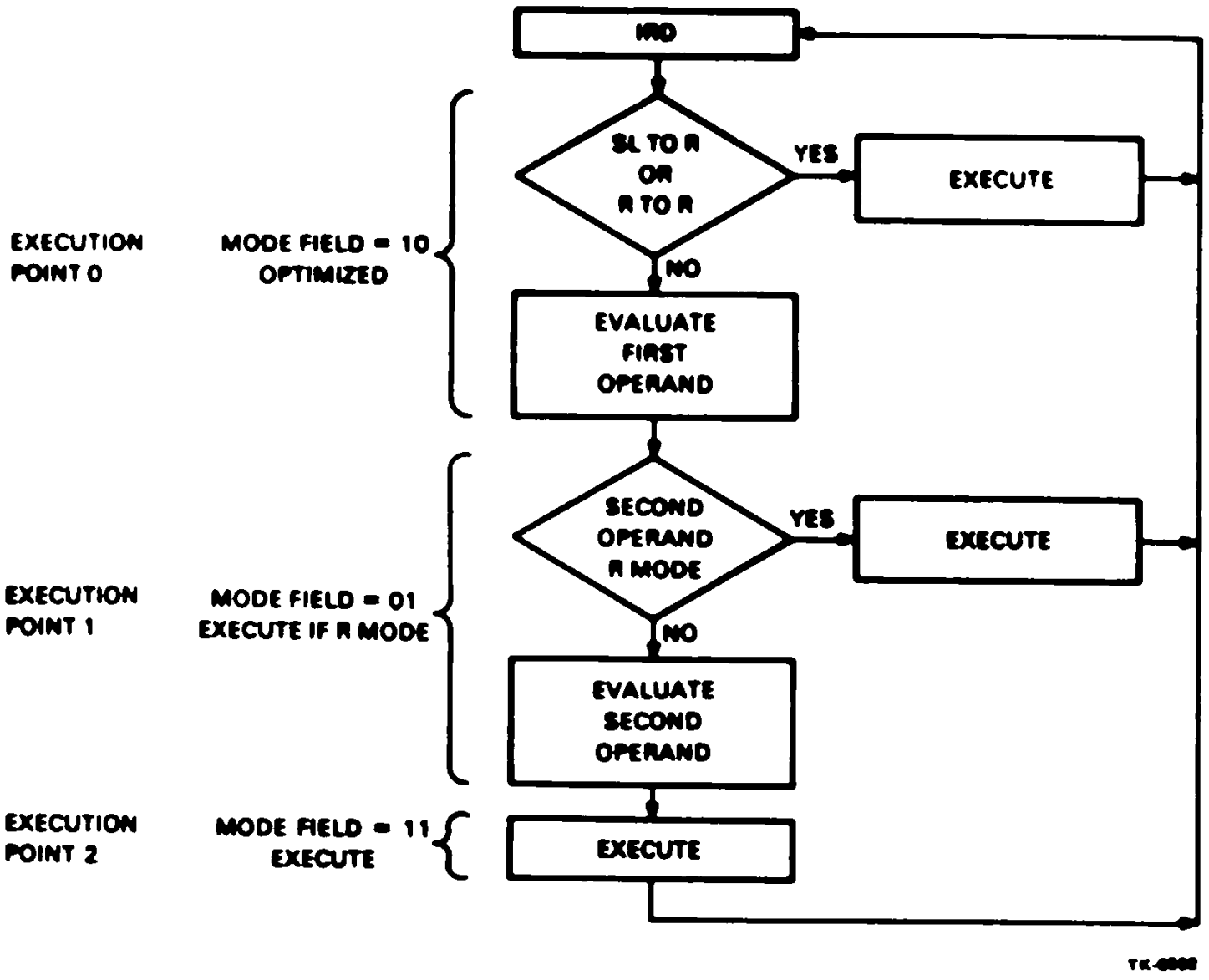


Figure 3-30 Double Operand Optimizing Flowchart

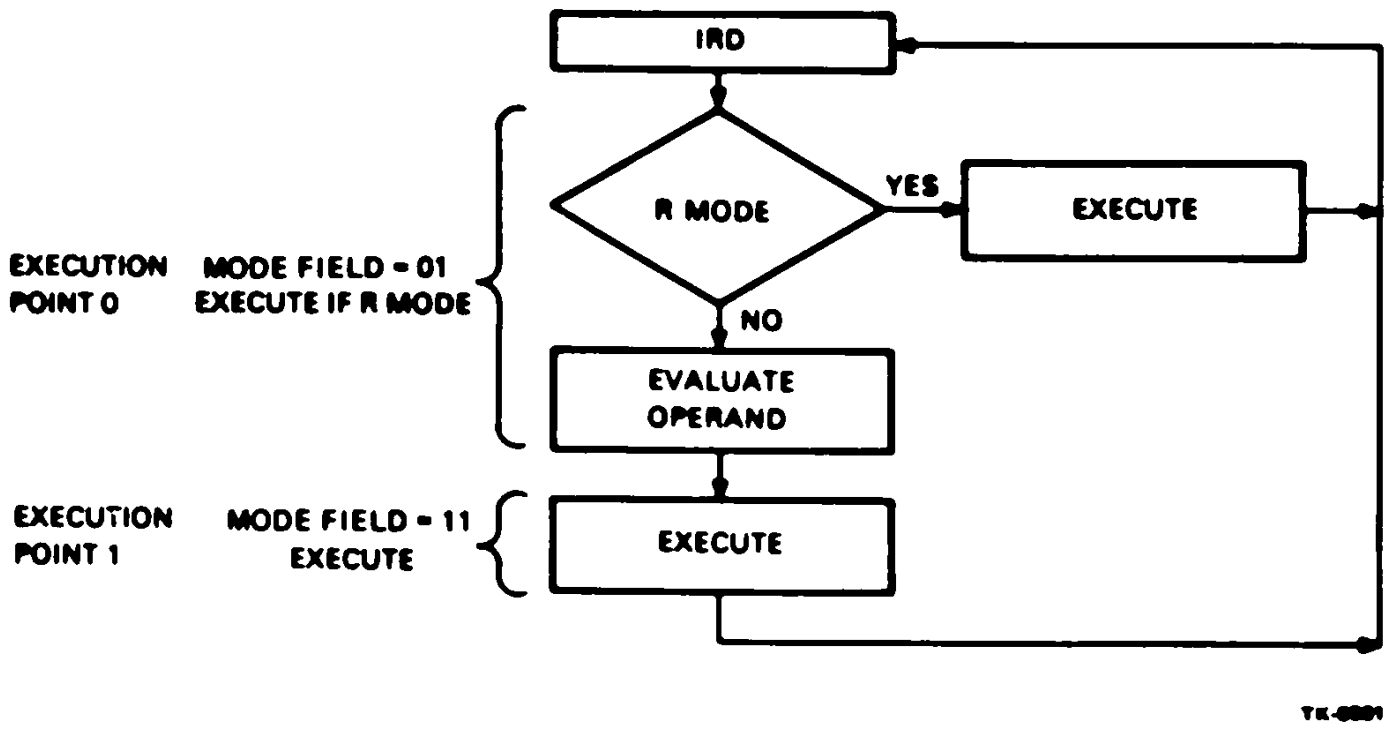


Figure 3-31 Single Operand Optimizing Flowchart

3.3.2.6 Branch Decode Logic – When a VAX-11 or PDP-11 branch instruction is decoded, the branch condition bits (BR COND ⟨02:00⟩) shown in Figure 3-32 are inclusive-ORed with the VAX DECODE or CM DECODE bits that form the microaddress. In native mode, the four low-order bits of the opcode (IR ⟨03:00⟩) are asserted to the branch decode logic to determine the branch condition. In compatibility mode, bits from the upper half of the PDP-11 instruction (BUF B1 ⟨07,02:00⟩) are decoded to identify the instruction. The condition code bits (N, Z, V, C) of the processor status longword (PSL) are asserted to the branch decode logic and combined with the instruction lines to form the branch condition bits (BR COND ⟨02:00⟩). The microaddress is a function of the operating mode (native or compatibility), the branch instruction being executed, and the status of the PSL condition code bits.

3.3.2.7 Size Select – The size select logic shown in Figure 3-33 generates the PC update value (described in Section 3.3.1.3) and the STALL condition.

In certain addressing modes, the operand specifier requires more data to generate the operand address. When these modes are encountered, the PC is incremented to reflect the length of the additional data (pointing beyond the operand specifier and its extension).

The STALL condition is also generated in these addressing modes and the number of required bytes are not valid. The STALL condition inhibits the execution point counter from being incremented and forces the microprogram to try another call. This continues until all necessary bytes are valid in the buffer register, preventing the microcode from moving to the next decision point before the specifier is evaluated.

In both native and compatibility mode, specifiers are decoded to check for the addressing modes that require additional data. The number of additional bytes required for each mode is shown in Table 3-24.

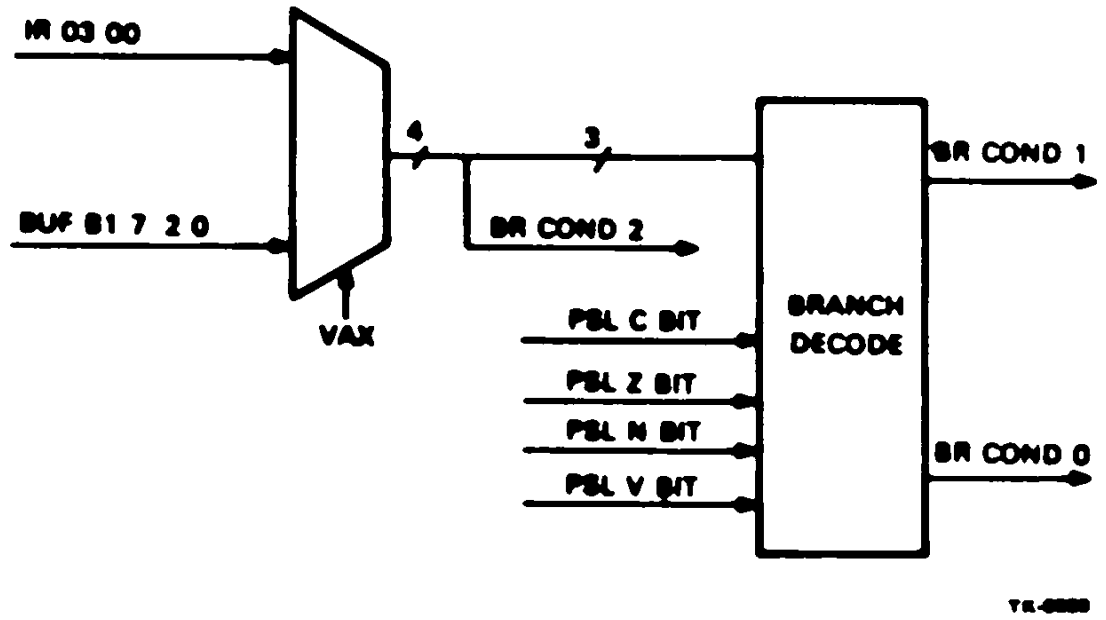


Figure 3-32 Branch Decode Logic

Table 3-24 Size Selection for PC Updates

Addressing Modes	Additional Bytes Required
In Native Mode	
Displacement or Displacement Deferred	
Byte	1
Word	2
Longword	4
Immediate	1, 2, or 4 bytes, depending on context
Absolute	4
Branch Displacement	
Byte	1
Word	2
In Compatibility Mode	
Index	2
Index Deferred	2
Immediate	2
Absolute	2

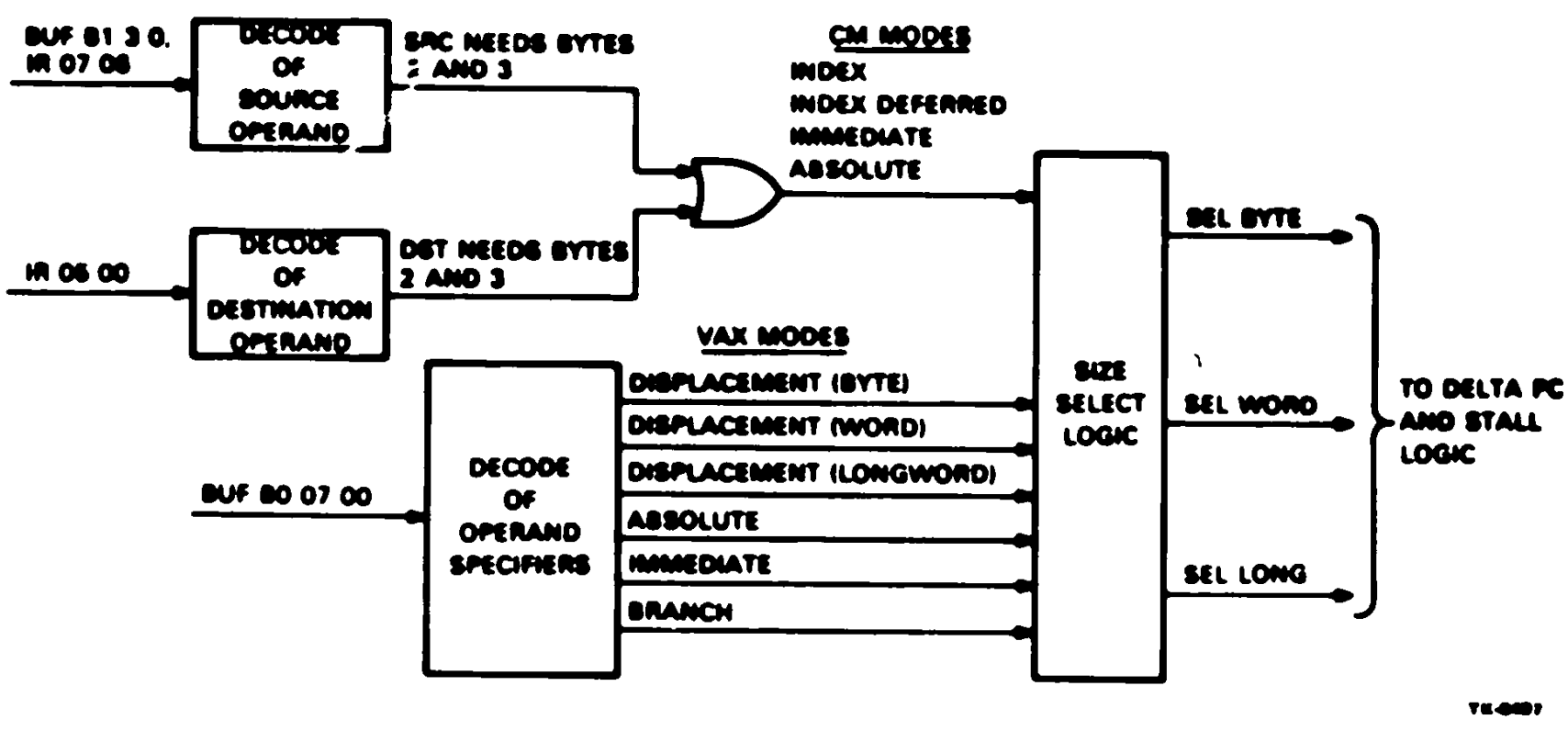


Figure 3-33 Size Select Logic

3.3.2.8 Index (I) Mode Flag – In index (I) mode, the operand specifier consists of two bytes; a primary operand specifier and a base operand specifier as shown in Figure 3-34.

When the addressing mode is decoded as I mode, incrementing of the execution point counter is inhibited when the primary operand specifier is removed from byte 1 of the buffer register. This prevents the microcode from moving to the next decision point before both specifiers are evaluated and the operand address generated.

The I mode flag is set as shown in Figure 3-35 when the primary operand specifier is in I mode and has been cleared from the buffer register. The flag allows the hardware to evaluate the base operand specifier and to retain the mode of the primary specifier. An abort address is generated (as described in Section 3.3.2.4) if the base operand specifier is in register, literal, or index mode, or if the PC is used as the index register in the primary operand specifier. The I mode flag is cleared when the opcode is cleared from byte 0 of the buffer register or when the base operand specifier is cleared from the buffer.

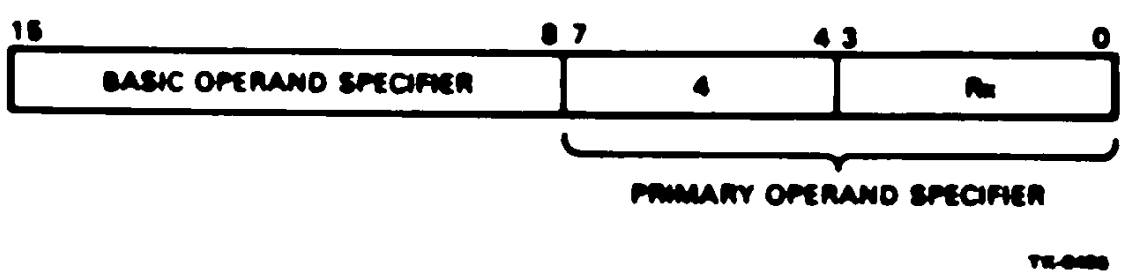


Figure 3-34 Index (I) Mode Operand Specifiers

3.3.2.9 Specifier Constants – The specifier constants, generated by the instruction decode logic shown in Figure 3-36, are input to the fast constant multiplexer of the CPU data paths. These constants are generally used in the evaluation of the autoincrement and autodecrement modes.

In native mode, the specifier 1 constant (SP1 CON ⟨03:00⟩) is a value determined by the data type of the operand specifier under evaluation; 8 (quadword), 4 (longword), 2 (word), and 1 (byte). Specifier 2 constant (SP2 CON ⟨02:01⟩) is 0.

In compatibility mode, the specifier 1 constant is 1 or 2, determined by the data type (byte or word) of the instruction and the number of the source register. Specifier 2 constant is the number 1 or 2, determined by the instruction data type and the number of the destination register.

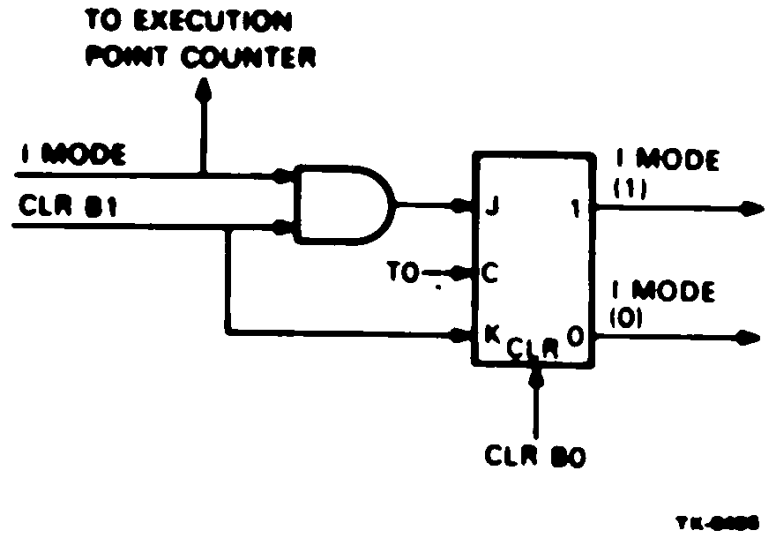


Figure 3-35 Index (I) Mode Flag Logic

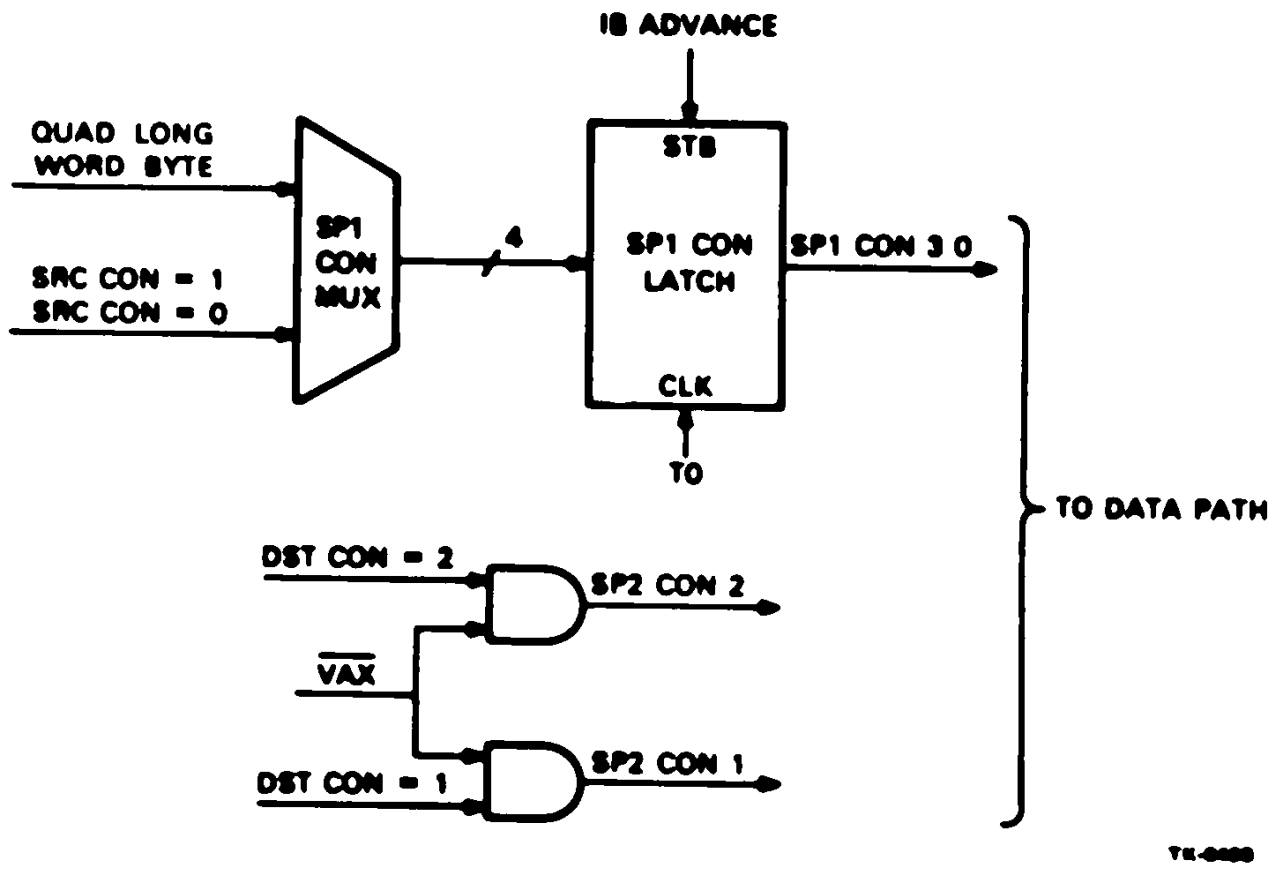


Figure 3-36 Specifier Constants Logic

3.3.2.10 Microsequencer Branch Conditions - The instruction decode logic provides two branch condition bits (BRC (01:00)), used by the microsequencer (USC) to form microbranch addresses. These condition bits are transferred from the branch condition multiplexer shown in Figure 3-37 to a branch multiplexer on the interrupt control (ICL) module. When the UBEN field of the current microword equals 8, 9, A, or B, in native or compatibility mode, the BRC bits select the two low-order bits of the next microaddress according to the conditions shown in Table 3-25.

3.3.2.11 Compatibility Mode Decoding - Compatibility mode is specified when bit (31) (CM bit) of the PSL is set. This mode enables execution of the PDP-11 instruction stored in bytes 0 and 1 of the instruction buffer register in one of the formats shown in Figure 3-38.

The compatibility mode decode logic asserts address lines CM DECODE (07:00) to the mode multiplexer to assert the next microaddress to the UPC ADRS MUX as shown in Figure 3-25. The PDP-11 execution ROMs and CM decode logic shown in Figure 3-39 then direct the microcode to a flow that either executes the instruction or evaluates the source or destination mode of the operand.

PDP-11 Execution ROM Decoding - Compatibility mode decode bits are generated from the PDP-11 execution ROMs or from a decode of the source/destination mode field of the instruction. The execution ROMs are divided between double operand and register class instructions, single operand instructions, and single byte instructions, one of which is enabled by the format of the PDP-11 instruction stored in buffer register bytes 0 and 1. If byte 1 is all zeros, for example, the single byte ROM is enabled and the outputs are asserted to the CM DECODE multiplexers. The execution lines are then asserted to the mode mux when compatibility mode execute (CM EXEC) is enabled.

Assertion of the CM EXEC line depends on the instruction class and the state of the execution point counter bits EXC CT (01:00). The execution ROMs generate the microaddress when all necessary source and destination operand evaluations have completed. The conditions under which CM EXEC is generated for specific execution point counter values are shown in Table 3-26.

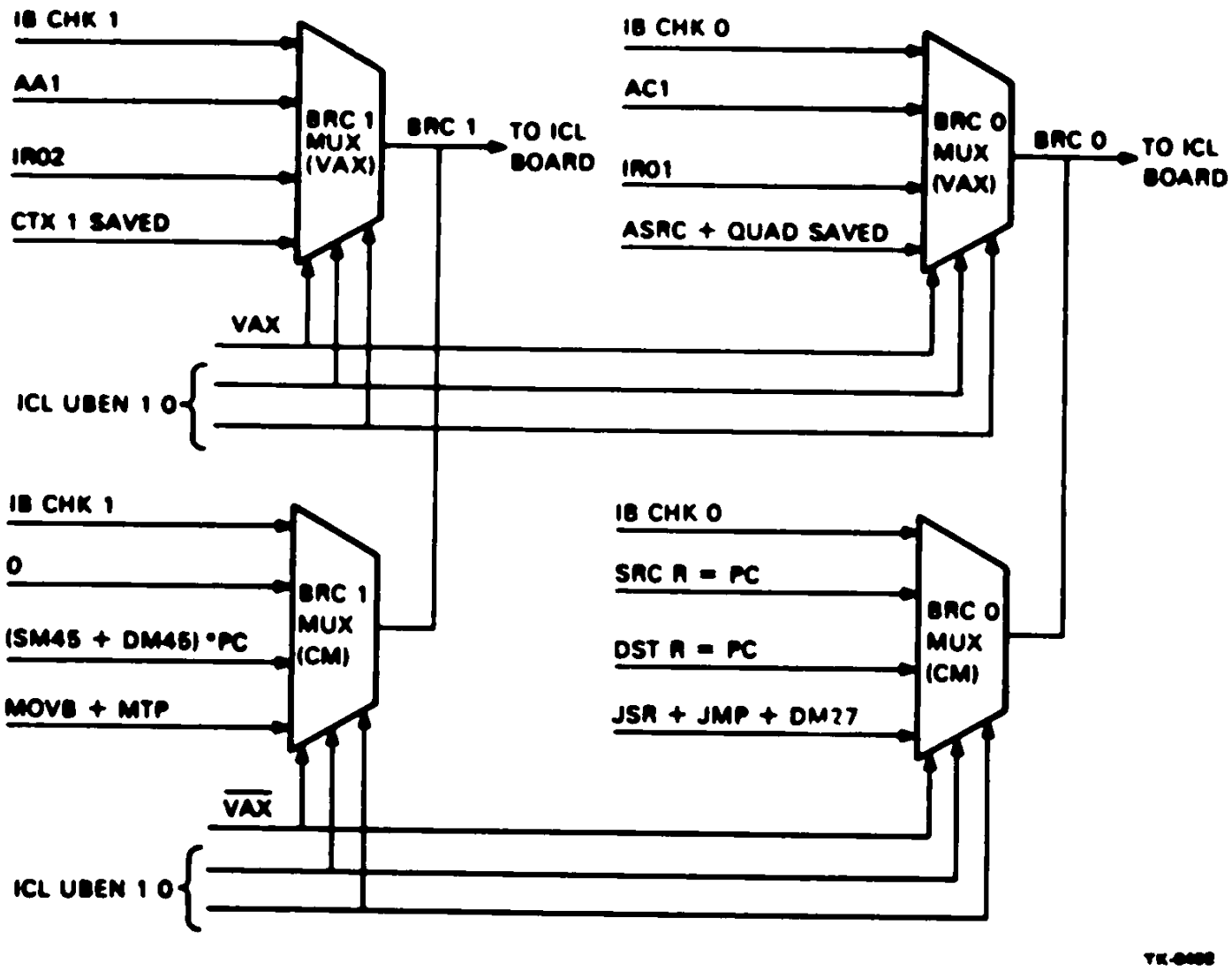


Figure 3-37 Microbranch Condition Multiplexers

Table 3-25 UBEN Field Control of the ICL Branch Multiplexer

UBEN Value	Native Mode BRC ⟨01⟩	BRC ⟨00⟩	Compatibility Mode BRC ⟨01⟩	BRC ⟨00⟩
8	ASRC or VSRC 0 = Normal 1 = Quad 2 = Field Source 3 = Address Source	ASRC or QUAD	0 Class	J class or DM27
9	IR⟨2⟩	IR⟨1⟩	SM or DM = 47 or 57	DST R = PC
A	-	-	0	SRC R = PC
B	IB CHK ⟨1⟩ 0 = TB Miss 1 = Error 2 = STALL, 3 = Data OK	IB CHK ⟨0⟩	IB CHK ⟨1⟩ 0 = TB Miss 1 = Error 2 = STALL 3 = Data OK	IB CHK ⟨0⟩

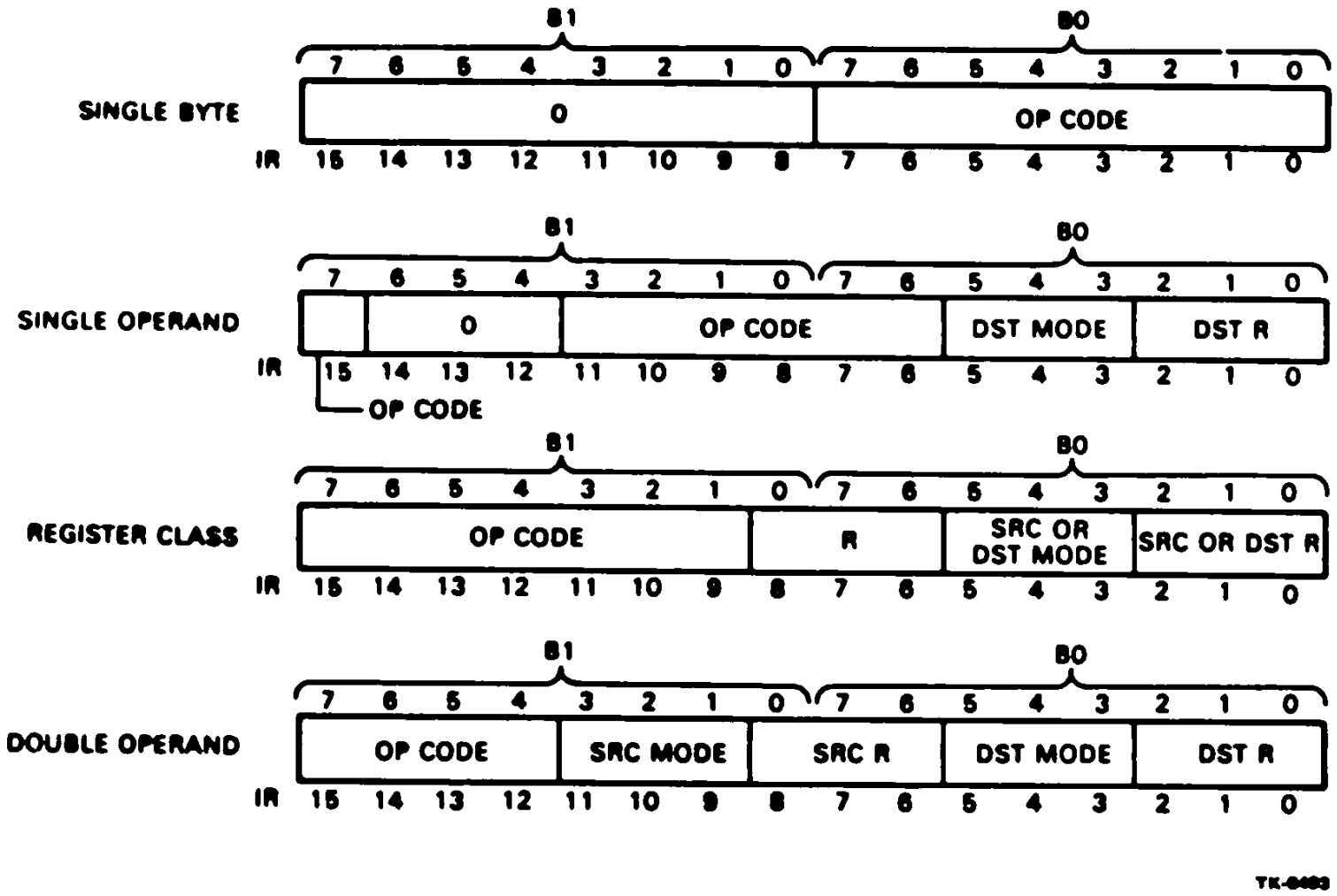


Figure 3-38 PDP-11 Instruction Buffer Register Format

Table 3-26 Execution Point Counter Generation of CM EXEC

Execution Point Counter EXC CT ⟨01:00⟩ Values	Conditions that Assert CM EXEC
EXC CT ⟨01:00⟩ = 00	Single byte OR DM = 0 AND SM = 0 OR DM = 0 AND register class OR DM = 0 AND single operand
EXC CT ⟨01:00⟩ = 01	SM = 0 OR DM = 0 OR register class OR single operand
EXC CT ⟨01:00⟩ = 10 or 11	Any combination of instruction class and operand mode

A maximum of three execution points are required to evaluate and execute any PDP-11 instruction. In double operand instructions with the source and destination modes not equal to zero, the source operand is evaluated at execution point 0, the destination operand is evaluated at execution point 1, and the instruction is executed at execution point 2. The instruction can be executed at execution point 0 if, for example, it is of a type with no operands, or is a double operand instruction with both operands in register mode.

Source/Destination Mode Decoding – When source or destination operand evaluations are required, the CM EXEC line is not asserted. The CM DECODE multiplexers select inputs from a decode of the instruction source or destination fields. The four lower bits, CM DECODE (03:00), are asserted generated from the source/destination (SRC/DST) multiplexer.

The source inputs to the multiplexer are enabled only under one set of circumstances; if the instruction is of double operand (binary) class, the execution point counter equals 0, and the source mode is not equal to zero. The destination inputs are enabled to evaluate the source or destination mode in register class instructions and to evaluate the destination mode in single or double operand instructions.

The four upper address bits CM DECODE (07:04) are generated from a decode of the SRC/DST mode fields and the op code. The execution of certain double operand instructions can be optimized if the operation is a literal to register transfer. Optimizations require that the source operand be in immediate mode and the destination operand be in register mode. The constant or literal data is then located in the second word of the instruction (buffer register bytes 2 and 3). The microaddress generated depends on the instruction opcode and the optimization code generated as shown in Figure 3-39.

3.4 CPU DATA PATHS

Figure 3-40 is a block diagram of the CPU data path logic that consists of the five modules M7467 through M7471 in slots 9 through 13. The data paths are divided into the following four major sections of logic:

- Address section
- Data section
- Arithmetic section
- Exponent section

Each section operates as an independent unit, processing address or data information in parallel with the operations taking place in the other sections.

Under control of the CPU microprogram, the data paths perform the sequential operations that execute VAX-11 or PDP-11 instructions. Each section functions under the control of a microword bit field from the control store (CS) bus as shown in Figure 3-41.

3.4.1 Address Section

The address section contains the instruction buffer register control logic. It makes use of the following registers that are loaded through but are incremented independently of the arithmetic section:

- Virtual instruction buffer address (VIBA) register for virtual memory instruction fetches
- Virtual address (VA) register for virtual memory data references
- Program counter (PC) register for CPU program control and sequencing

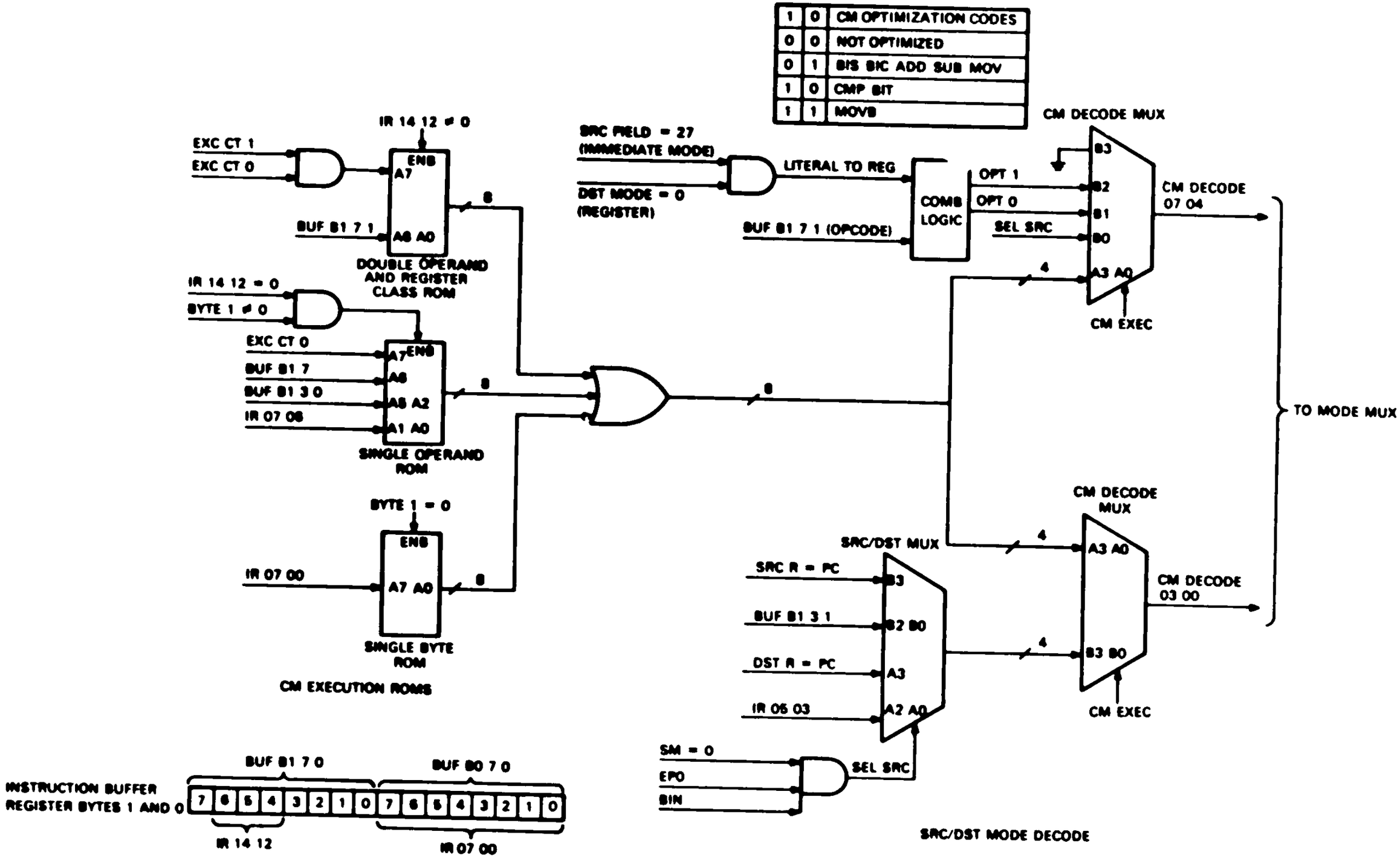


Figure 3 39 Compatibility Mode Decode Logic

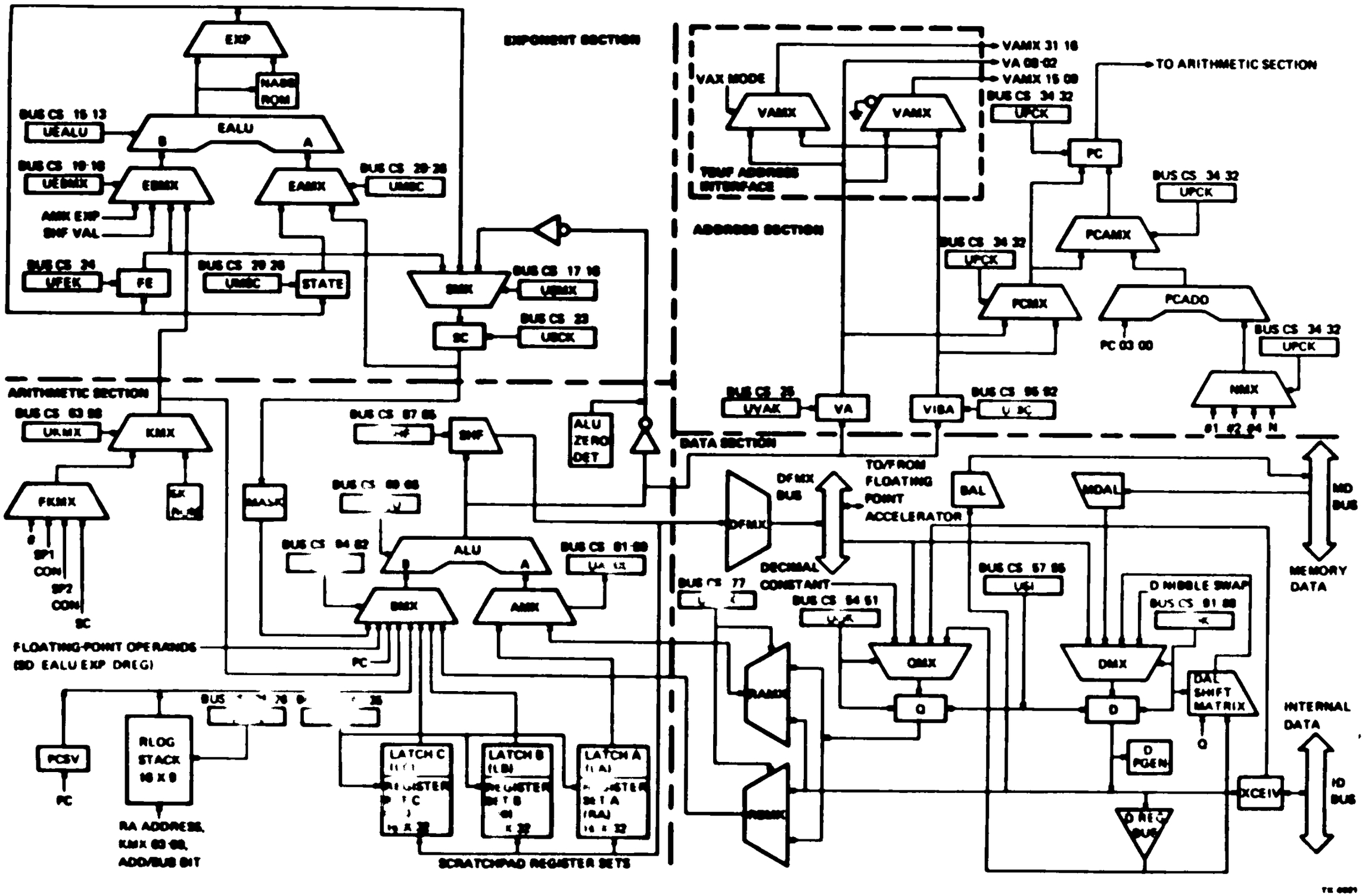


Figure 3-40 CPU Data Path Block Diagram

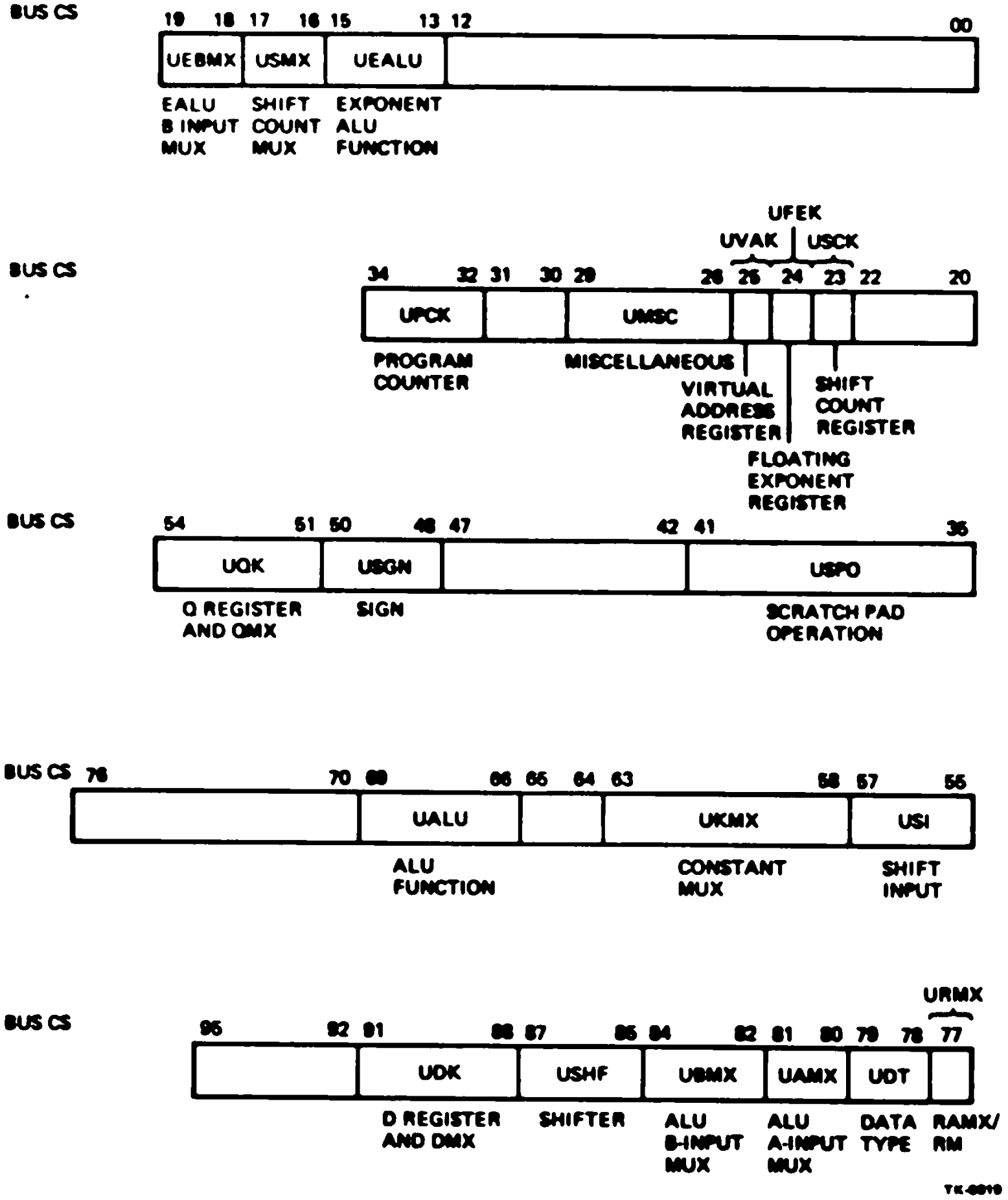


Figure 3-41 Microword Fields Used for Data Path Control

3.4.1.1 Virtual Instruction Buffer Address (VIBA) Register – The VIBA register is part of the addressing logic shown in Figure 3-42.

The VIBA holds the virtual address of the I-stream data to be fetched from memory by the instruction buffer control logic and stored in the instruction buffer register on the IDP module. The virtual address in the VIBA is then converted to a physical memory address by a memory translation process using the translation buffer. The two lowest bits (VIBA ⟨01:00⟩) are not part of the translation address. These bits control byte rotation of the I-stream data loaded to the instruction buffer register on the IDP module. This rotation aligns the opcode with the instruction execution ROMs on the IRC module.

The VIBA is controlled by the UIBC field of the microinstruction. When the UIBC field equals 2, the VIBA register is loaded with data from the arithmetic section (ALU (31:02)).

The VIBA is loaded with a new address on the program sequence change that takes place on a JUMP or successful BRANCH instruction, or on the initialization of a trap or interrupt routine. When the new instruction data (the longword containing the opcode) has been successfully fetched, the instruction buffer control logic increments the VIBA by four.

3.4.1.2 Virtual Address (VA) Register – The VA register is part of the addressing logic shown in Figure 3-42. The VA register holds the virtual address of an instruction reference that transfers data between memory and the data section. It normally contains a virtual address that is converted to a physical memory address from the translation buffer. However, it may also store a physical address generated by the translation process or when the memory management mechanism is disabled.

The VA is used as an index to the translation buffer when the translation buffer is being updated or invalidated by the microprogram. The index function is also used by the PROBE instruction to determine if an access violation might occur if a memory reference is made to the specified virtual address.

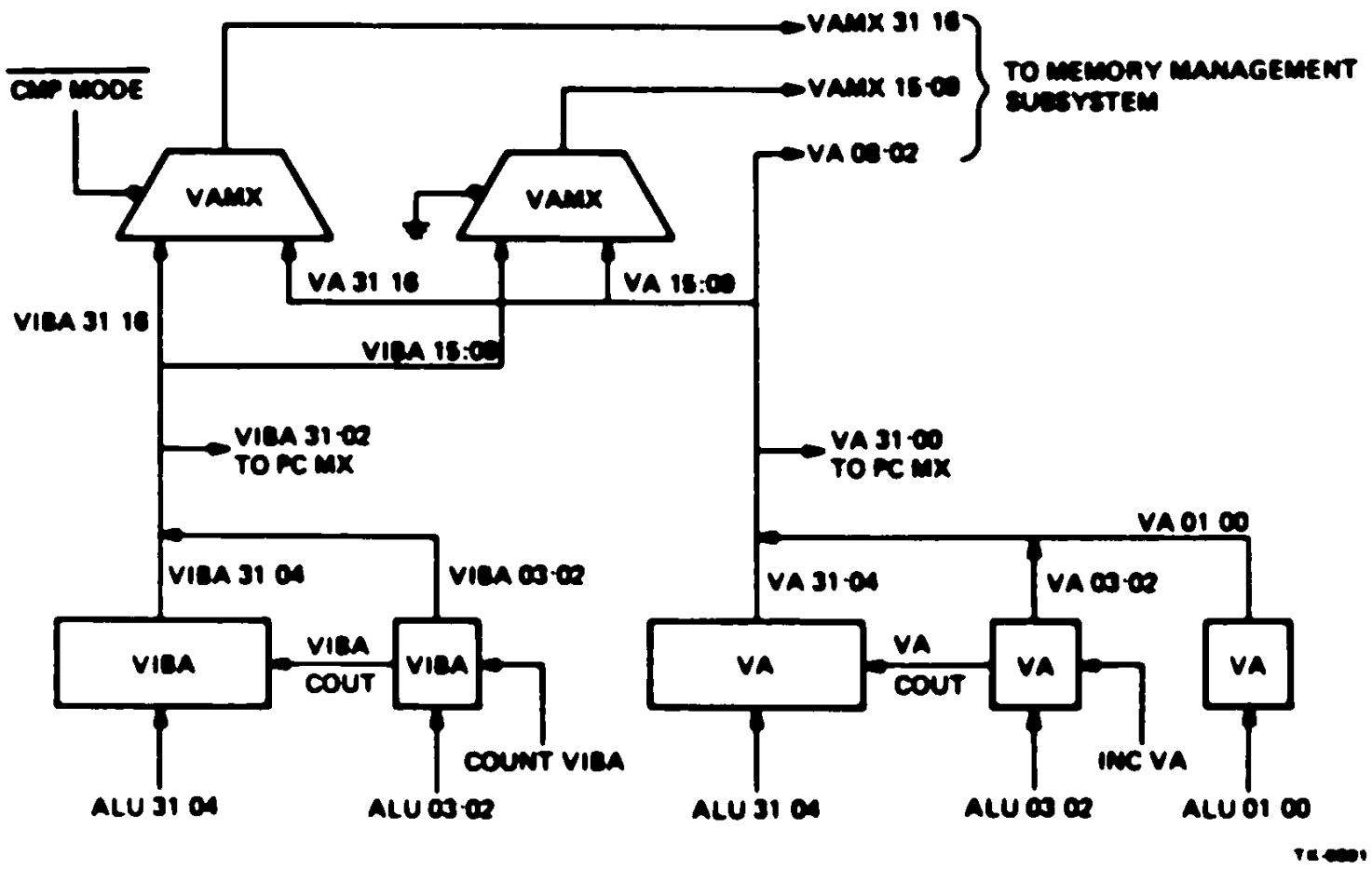


Figure 3-42 VIBA and VA Registers and Multiplexers

The VA is controlled by the one-bit UVAK field of the microinstruction as shown in Table 3-27.

The VA is incremented by four when the three-bit UPCK field of the microinstruction equals 3. However, the UVAK load function overrides the increment if both functions are selected.

Table 3-27 UVAK Bit Control of the VA Register

UVAK Bit	Function
0	Hold
1	Load VA with ALU (31:00) from arithmetic section

3.4.1.3 Virtual Address Multiplexer (VAMX) – The VAMX provides the interface to the memory management subsystem (translation buffer and cache) and provides the correct virtual address format for either native mode or compatibility mode as shown in Figure 3-43.

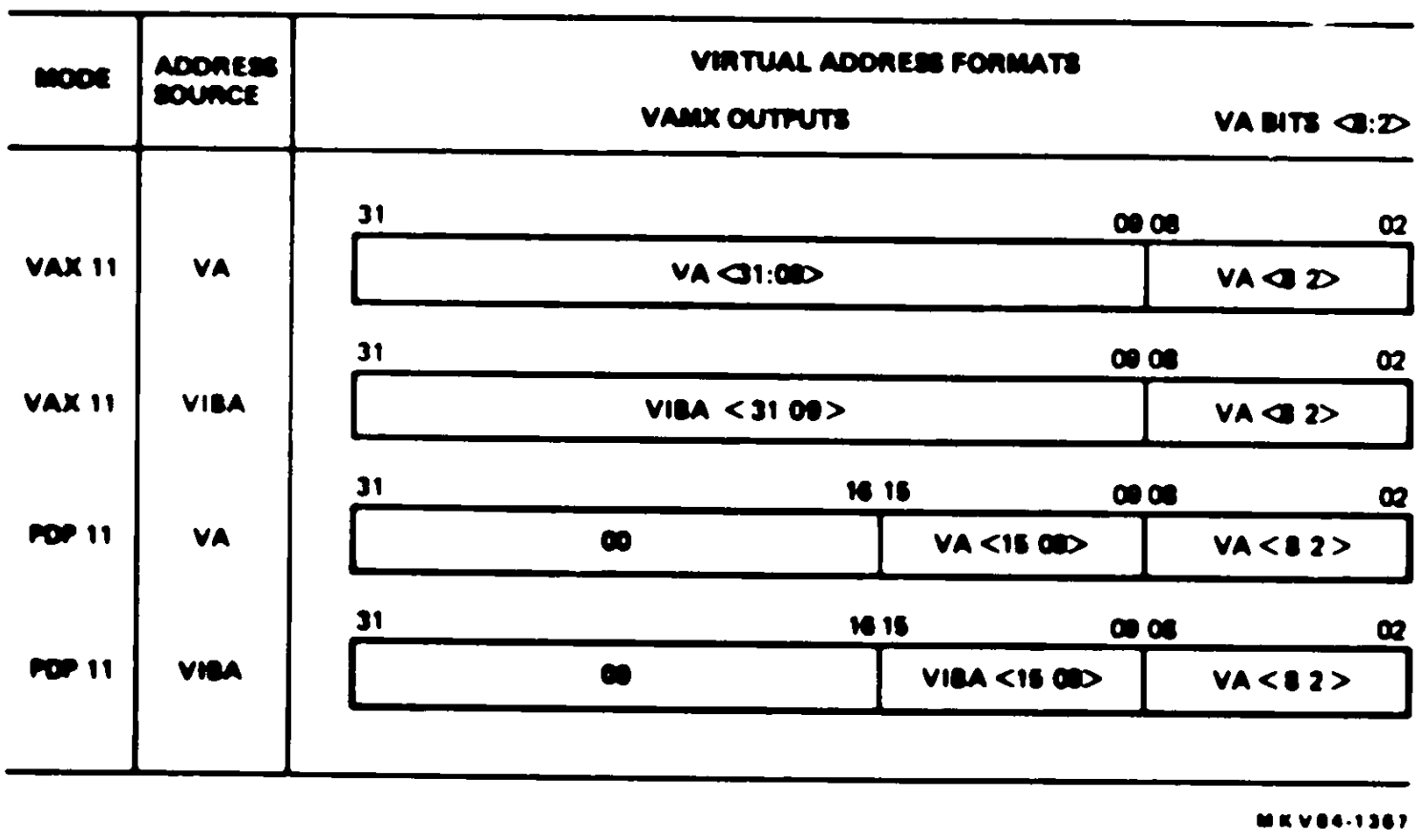


Figure 3-43 VAMX Data Formats

The state of the compatibility mode (CM) bit in the PSL determines which set of VAMX formats is selected The VA or VIBA is then selected as the source for address bits (31:09). Address bits (08:02) are taken from the VA and are not input to the VAMX. Address bits (01:00) are not sent to the memory management subsystem because all memory references are made on longword boundaries and these address bits select the position of a byte in the longword.

The address source is selected by a signal from the translation buffer. The VA is selected when the microprogram requests a memory data reference The VIBA is selected when the microprogram requests I-stream data and the instruction buffer control logic may use the same cycle to request a memory transfer.

3.4.1.4 Program Counter (PC) Register - The PC register shown in Figure 3-44 holds the address of the instruction opcode for the start of the next execution sequence. It is then incremented by the correct number of bytes used as the instruction opcode and operand specifiers are evaluated.

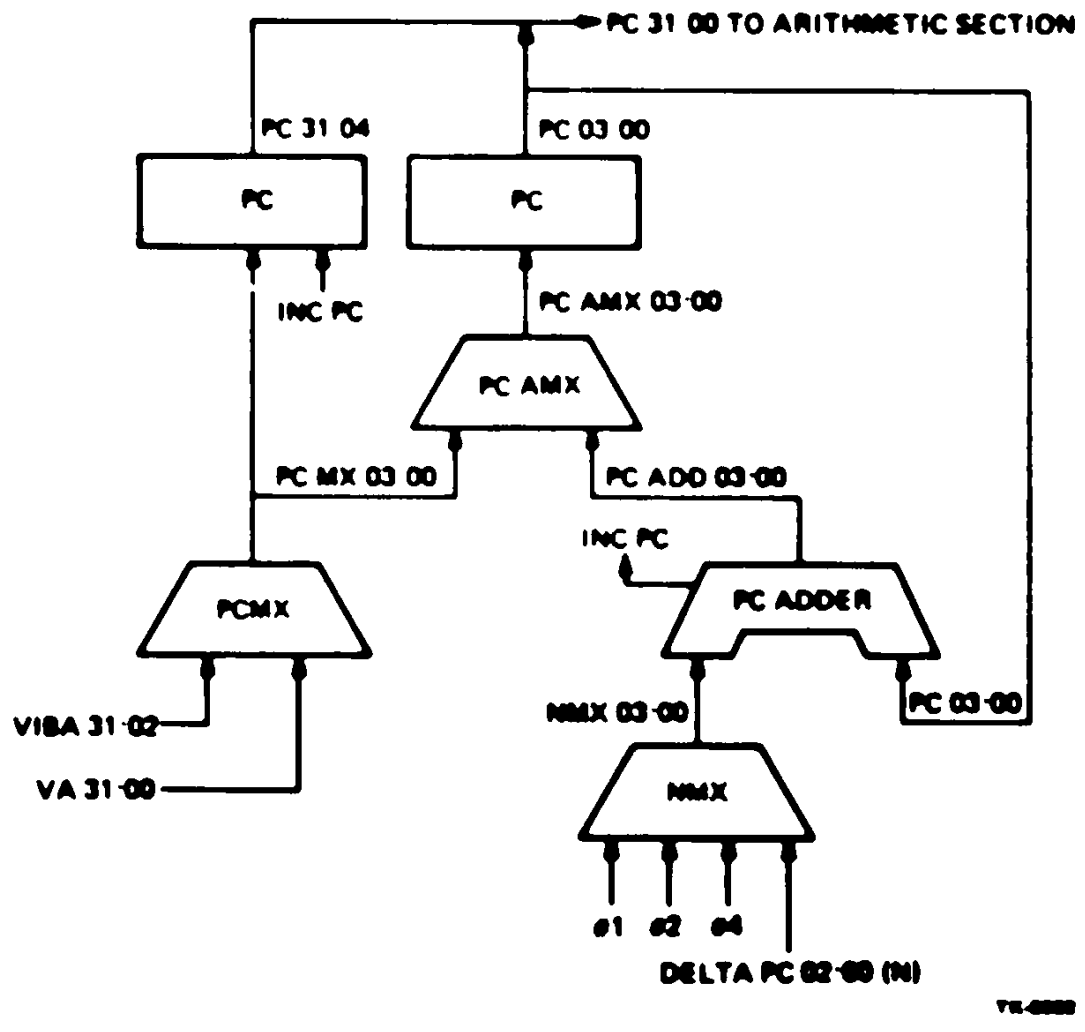


Figure 3-44 Program Counter (PC) Register and Multiplexers

The PC is loaded from the VIBA or VA register on a program sequence change such as for a JUMP or BRANCH or a deferred JUMP or BRANCH instruction. The data source for PC bits (31:04) is either the VIBA or VA register from the PCMX. The data source for PC bits (03:00) is either the VIBA or VA register or PC ADDER from the PCAMX.

The PC ADDER allows the PC to be incremented by 1, 2, 4, or n. The instruction buffer control logic generates the value n, incrementing the PC to a point beyond the I-stream bytes being evaluated.

The PC input selection is controlled by the three-bit UPCK field of the microinstruction as shown in Table 3-28.

3.4.1.5 PC Adder (PCADD) and Number Multiplexer (NMX) - The PCADD performs the function "PC (03:00) plus NMX (03:00)". The numbers 1, 2, 4, or n are selected by the NMX to provide the increment value. The value n may be 1, 2, or 4 and is generated by the instruction buffer decode logic on the IRC module (as described in Section 3.3.1.3).

The outputs of the PCADD (PC ADD (03:00)) are multiplexed by the PC adder multiplexer (PC AMX) which provides the inputs to PC bits (03:00). If the PC adder function results in a carry, the upper 28 bits of the PC are also incremented. The increment value selected by the NMX is controlled by the UPCK field of the microinstruction (as described in Section 3.4.1.4).

Table 3-28 I PCK Field Control of the PC

I PCK Value	Function Selected
0	NOP
1	VA input to PC
2	VIBA input to PC
3	Increment VA by 4
4	Increment PC by 1
5	Increment PC by 2
6	Increment PC by 4
7	Increment PC by n

3.4.1.6 PC Multiplexer (PCMX) and PC Adder Multiplexer (PCAMX) - The PCMX and PCAMX provide the data inputs to the PC When the PC is first loaded, the PCMX selects the VIBA or VA as the input and the PCAMX selects PCMX (03:00) All PCMX bits (VIBA (31:02) or VA (31:00)) are then loaded to the PC at the beginning of the instruction sequence

When the PC is incremented, the PCAMX selects PC ADD (03:00)as the inputs for the lower four bits of the PC The inputs to the upper bits are disabled and their value is unchanged until an increment generates a carry from the lower bits

3.4.2 Data Section

The data section handles data transfers between memory and the internal CPU registers It performs data shifting and byte alignment and the unpacking of floating-point data

The data section contains the Q and D registers These are the two main 32-bit registers used for the temporary storage of operand data and are used together to handle data types larger than 32 bits.

3.4.2.1 Data Section Interface - The data section logic shown in Figure 3-45 interfaces with the arithmetic section and the internal data (ID) bus through the data format multiplexer (DFMX) and through the register A and B multiplexers (RAMX and RBMX)

Data Format Multiplexer (DFMX) - The DFMX transfers data from the arithmetic shifter (SHF (31:00)) in either integer or unpacked floating-point format During execution of a floating-point instruction, shifter data is in the packed floating-point format shown in Figure 3-46 The DFMX unpacks the floating-point format by reassembling the fraction and removing the sign and exponent bits as shown in Figure 3-47

If the integer format is selected, shifter data passes through the DFMX unmodified. DFMX format selection is controlled by the Q control (UQK) and D control (UDK) fields of the CPU microword When either field equals 8, integer format is selected When either field equals 9, unpacked floating-point format is selected If the UDK field equals A or the UQK field equals B, floating point accelerator (FPA) data is transferred to the D or Q register multiplexer (DMX or QMX) on the DFMX bus.

Register A (RAMX) and Register B (RBMX) Multiplexers - The RAMX asserts either the Q or D register to the ALU A input multiplexer in the arithmetic section The RBMX asserts either the Q or D register to the ALU B input multiplexer. (RAMX and RBMX selection is described in Section 3 4.3 2 and Section 3.4.3.3.)

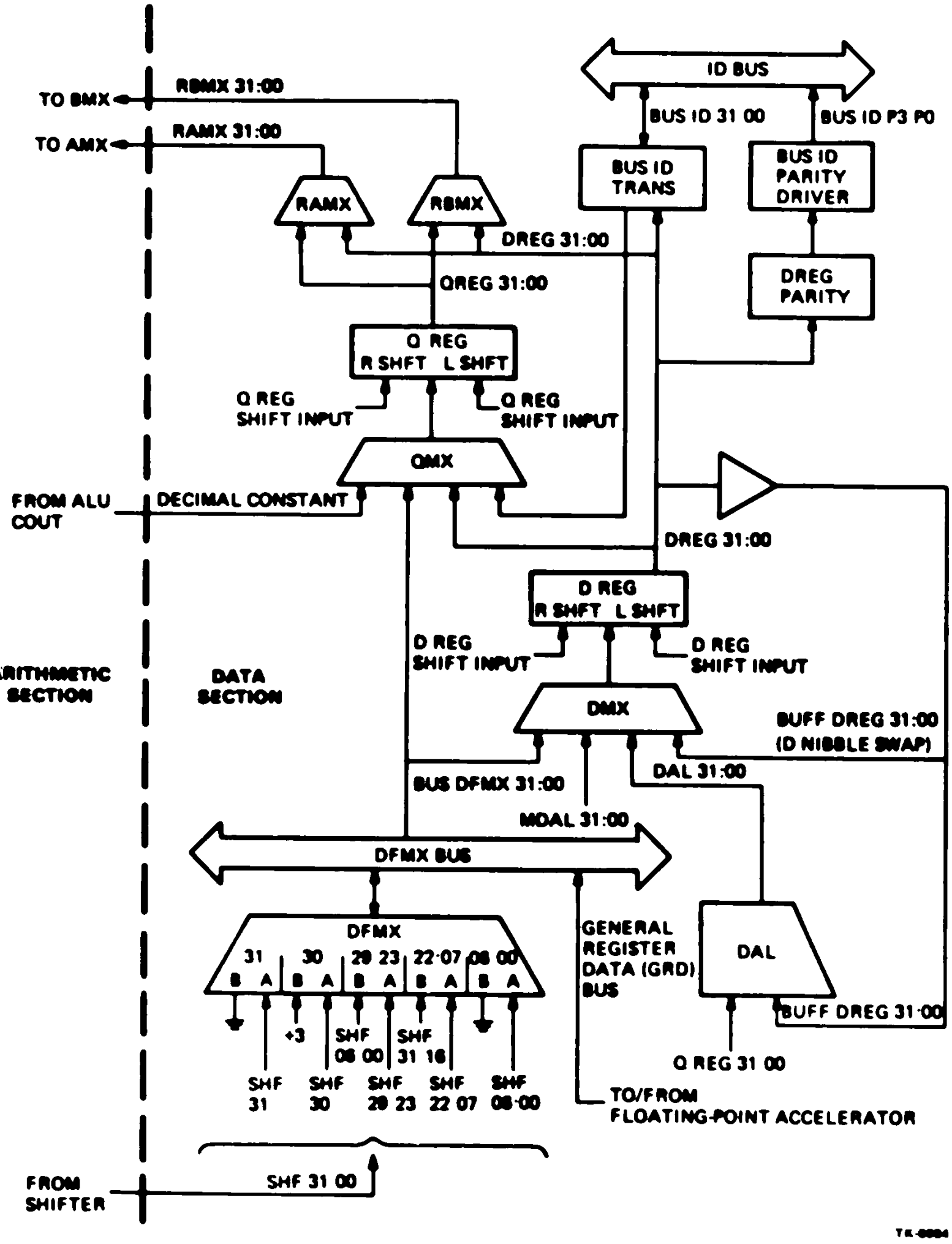


Figure 3-45 Data/Arithmetic Section Interface

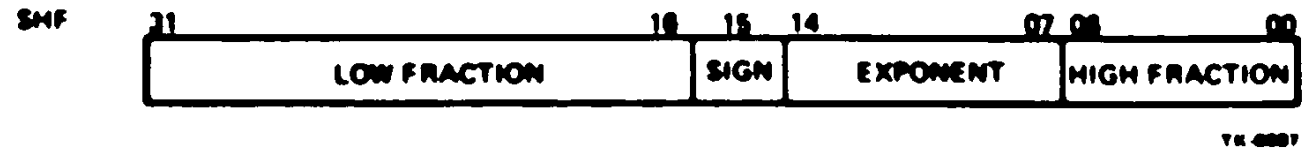


Figure 3-46 Packed Floating Point Format

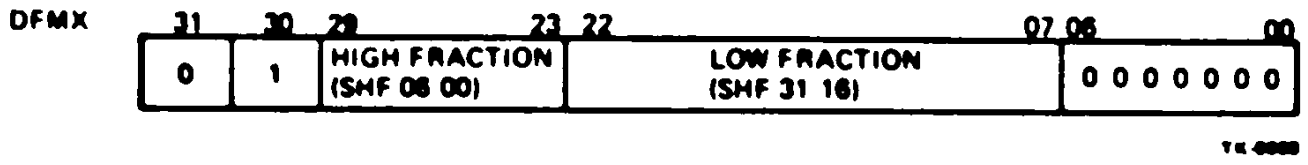


Figure 3-47 Unpacked Floating Point Format

3.4.2.2 Q and D Registers and Data Aligner (DAL) – The Q and D holding registers provide temporary storage for operand data generated by the other data path sections. The data aligner (DAL) performs the shifting for certain register functions.

Q Register – The Q register is used for the following functions:

- During the execution of field and double-floating-point instructions, the Q and D registers are used together to hold data types larger than 32 bits.
- During the execution of multiply and divide instructions, the Q register holds the multiplier and quotient.
- During the evaluation of instruction operands, the Q register holds the first operand while the next operand is evaluated.

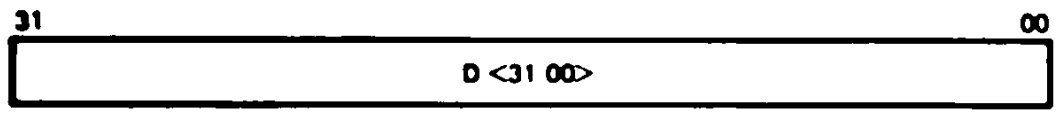
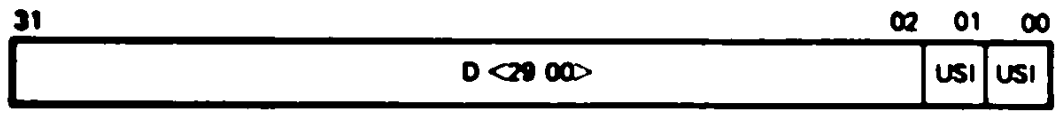
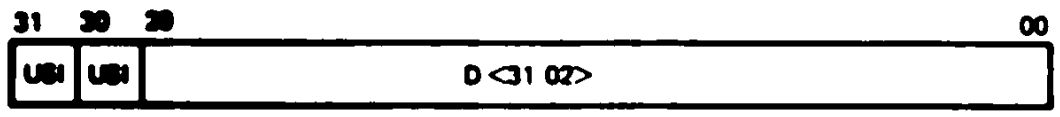
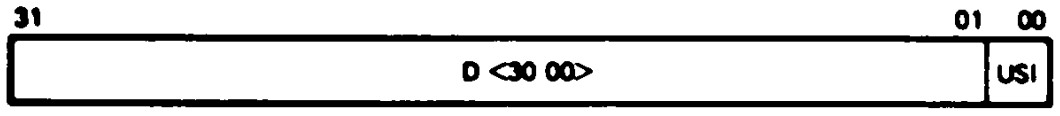
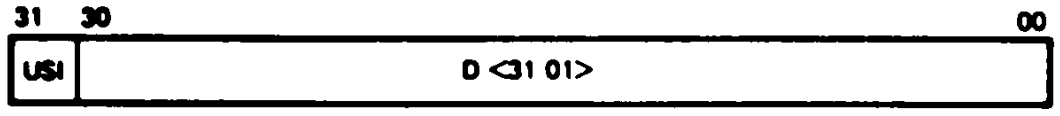
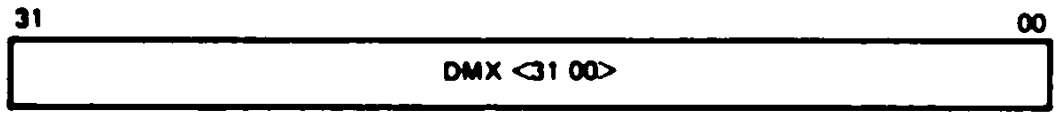
The Q register may be shifted right or left one or two bits or may be loaded from the the Q register multiplexer (QMX). The function is determined by the UQK field of the microinstruction as shown in Figure 3-48.

If the function is a Q register shift, the value shifted into the end bit is determined by the shift input field (USI) of the microword as shown in Table 3-29.

Q Register Multiplexer (QMX) – The QMX provides the data source for the Q register with data selected by the UQK field values shown in Figure 3-49. When the UQK field has a value of 8 through F, the QMX asserts the DFMX, the D register, the ID bus, or asserts a decimal constant of 6 (0110 binary) to each nibble. A constant of 6 is added to each nibble as an adjustment when performing binary coded decimal (BCD) arithmetic.

D Register – The D register provides temporary data storage for the following functions:

- Holds data received from the MD bus.
- Holds data to be transmitted to the ID bus. When the D register is used for ID bus write operations, odd parity is generated for each byte. Parity is not checked on data received from the ID bus.
- Holds data types larger than 32 bits when used with the Q register.

UQK FIELD					Q REGISTER DATA FORMAT	FUNCTION
HEX	BUS CS S4	S3	S2	S1		
0	0	0	0	0		HOLD
1	0	0	0	1		DOUBLE SHIFT LEFT
2	0	0	1	0		DOUBLE SHIFT RIGHT
3,4						RESERVED
5	0	1	0	1		SINGLE SHIFT LEFT
6	0	1	1	0		SINGLE SHIFT RIGHT
7	0	1	1	1		RESERVED
8-F						LOAD DMX DATA

MKV84 1378

Figure 3-48 UQK Field Control of the Q Register

Table 3-29 USI Field Control of the Q Register Shift Input

USI Field	Q Register Shift Input
0	ALU CARRY 31
1	Q register bit ⟨31⟩
2	D register bit ⟨31⟩
3	0
4	0
5	ALU CARRY 31
6	0
7	1

UQK FIELD					QMX DATA SELECTED
HEX	BUS CS				
	54	53	52	51	
8	1	0	0	0	31 … 00 DFMX <31:00> (INTEGER FORMAT)
9	1	0	0	1	31 … 00 DFMX <31:00> (UNPACKED FORMAT)
A*	1	0	1	0	31 28 27 24 23 20 19 16 15 12 11 08 07 04 03 00 — 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 1 0 | 0 1 0
B	1	0	1	1	31 … 00 FLOATING-POINT ACCELERATOR DATA
C	1	1	0	0	31 … 00 D <31:00>
D	1	1	0	1	31 … 00 RESERVED
E	1	1	1	0	31 … 00 BUS ID <D31.00>
F	1	1	1	1	31 … 00 ZEROES

*DECIMAL 6 IS INPUT TO EACH QMX BYTE ONLY IF AN ALU CARRY IS GENERATED

MKV84-1369

Figure 3-49 UQK Field Control of QMX Input Selection

The D register may be shifted right or left by one or two bits or loaded from the D register multiplexer (DMX), determined by the UDK field of the microinstruction as shown in Figure 3-50.

If the function is a D register shift, the value shifted into the end bit is determined by the shift input field (USI) of the microword as shown in Table 3-30.

If the USI field equals 6 or 7 and the D register is selected for a double shift, ALU bit ⟨00⟩ is shifted to the D register on the first shift and ALU bit ⟨01⟩ on the second. If a single shift is selected and USI equals 6 or 7, ALU bit ⟨01⟩ is shifted in.

Table 3-30 USI Field Control of the D Register Shift Input

USI Field	D Register Shift Input
0	Q register bit ⟨31⟩
1	Q register bit ⟨00⟩
2	0
3	0
4	0
5	Q register bit ⟨31⟩
6	ALU ⟨00⟩/ALU ⟨01⟩
7	ALU ⟨00⟩/ALU ⟨01⟩

D Register Multiplexer (DMX) – When the UDK field has a value of between 1 and 7, the D register selects shifted data as shown in Figure 3-50.

When the UDK field has a value of 0 or between 8 through F, the DMX sources data to the D register, selected by the UDK field values as shown in Figure 3-51.

For a value of 0, the memory data aligner (MDAL) is selected from the DMX. When the UDK field has a value of between 8 and B, the DMX asserts the DFMX (integer or unpacked format), FPA data, or the D register buffer for the decimal nibble swap function. For UDK field values of C through E, the DMX selects shift outputs from the data aligner (DAL).

When the UDK field equals B, the DMX selects the buffered D register data to perform a decimal nibble swap. This is required to perform decimal arithmetic because of the way decimal string numbers are stored in memory In Figure 3-52, for example, the decimal number +12345678 is stored in consecutive memory byte locations. However, the buffered, reformatted outputs of the D register are asserted to the DMX in a format that correctly represents the decimal number as shown in Figure 3-53.

Data Aligner (DAL) – The DAL right or left shifts D register contents a maximum of 32 bits and shifts the contents of the Q register into the vacated bit positions as shown in Figure 3-54. It is used during the execution of decimal arithmetic, bit field, shift, multiply, divide, and floating-point instructions. The shift operation is also used to isolate bit fields on a virtual to physical address translation.

The DAL has three levels of shifters with the lower levels sourced to the next higher level inputs as shown in Figure 3-55. The resulting output is a combination of the shifts performed at each level.

UDK FIELD					D REGISTER DATA FORMAT	FUNCTION
HEX	BUS CS 91	90	89	88		
0	0	0	0	0	31 … 00: D <31:00>	HOLD
1	0	0	0	1	31 … 02: D <29:00>; 01: USI; 00: USI	DOUBLE SHIFT LEFT
2	0	0	1	0	31: USI; 30: USI; 29 … 00: D <31:02>	DOUBLE SHIFT RIGHT
3	0	0	1	1		RESERVED
4	0	1	0	0	31 … 01: D <30:00>; 00: USI	SINGLE SHIFT LEFT*
5	0	1	0	1	31 … 01: D <30:00>; 00: USI	SINGLE SHIFT LEFT
6	0	1	1	0	31: USI; 30 … 00: D <31:01>	SINGLE SHIFT RIGHT
7	0	1	1	1		RESERVED
8-F					31 … 00: DMX <31:00>	LOAD DMX DATA

NOTE

SINGLE SHIFT LEFT IS SELECTED IF ALU CARRY 31 IS GENERATED
WITH NO ALU CARRY, THE DMX OUTPUT (DFMX DATA IN INTEGER FORM) IS LOADED

MKV84-1373

Figure 3-50 UDK Field Control of the D Register

UDK FIELD					DMX DATA SELECTED
HEX	BUS CS 91	90	89	88	
0	0	0	0	0	31 … 00: MDAL 31 00 (NOTE 1)
8	1	0	0	0	31 … 00: DFMX 31 00 (INTEGER FORMAT)
9	1	0	0	1	31 … 00: DFMX 31 00 (UNPACKED FORMAT)
A	1	0	1	0	31 … 00: FLOATING-POINT ACCELERATOR DATA
B	1	0	1	1	31 … 24: BUFF DREG 07 00 | 23 … 16: BUFF DREG 15 03 | 15 … 08: BUFF DREG 23 16 | 07 … 00: BUFF DREG 31 24
C	1	1	0	0	31 … 00: DAL 31 00 (Q REG 31-00 NOTE 2)
D	1	1	0	1	31 … 00: DAL 31.00 (SHIFTED BY SC 09, 04 00 NOTE 2)
E	1	1	1	0	31 … 00: DAL 31-00 (SHIFTED BY SHF VAL, NOTE 2)
F	1	1	1	1	31 … 00: ZEROES

NOTE 1

UDK FIELD EQUALS 0 IF THE SIGNAL MD TO D IS ASSERTED BY THE SBI CONTROL BOARD, THE DMX SELECTS THE MDAL AS THE DATA SOURCE FOR THE D REGISTER

NOTE 2

UDK FIELD EQUALS C THE DAL PERFORMS A 32 BIT DATA SHIFT THAT RESULTS IN Q REGISTER BITS 31 00 BEING ASSERTED ON ITS OUTPUTS

UDK FIELD EQUALS D THE DAL SHIFT IS DETERMINED BY SHIFT COUNT (SC) REGISTER BITS 9 AND 4 0

UDK FIELD EQUALS E THE DAL SHIFT IS DETERMINED BY A NUMBER (NORM SHF VAL) GENERATED TO NORMALIZE THE FRACTION IN FLOATING-POINT INSTRUCTIONS.

MKV84 1365

Figure 3-51 UDK Field Control of DMX Input Selection

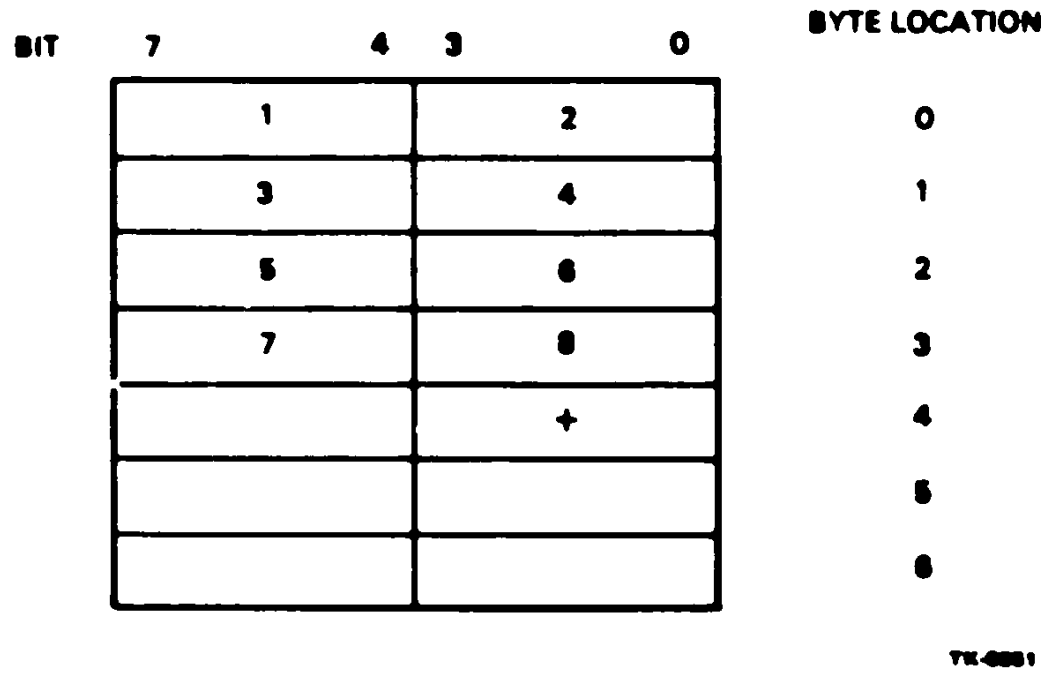


Figure 3-52 Decimal String Memory Format

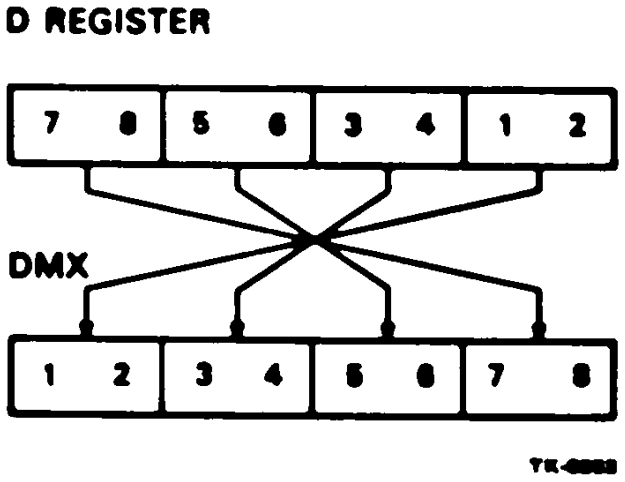


Figure 3-53 Swapped Decimal Number Format

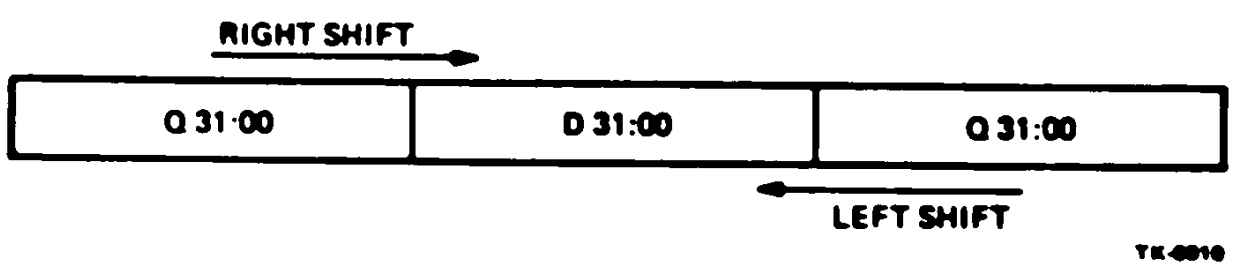


Figure 3-54 Data Aligner (DAL) Shift Formats

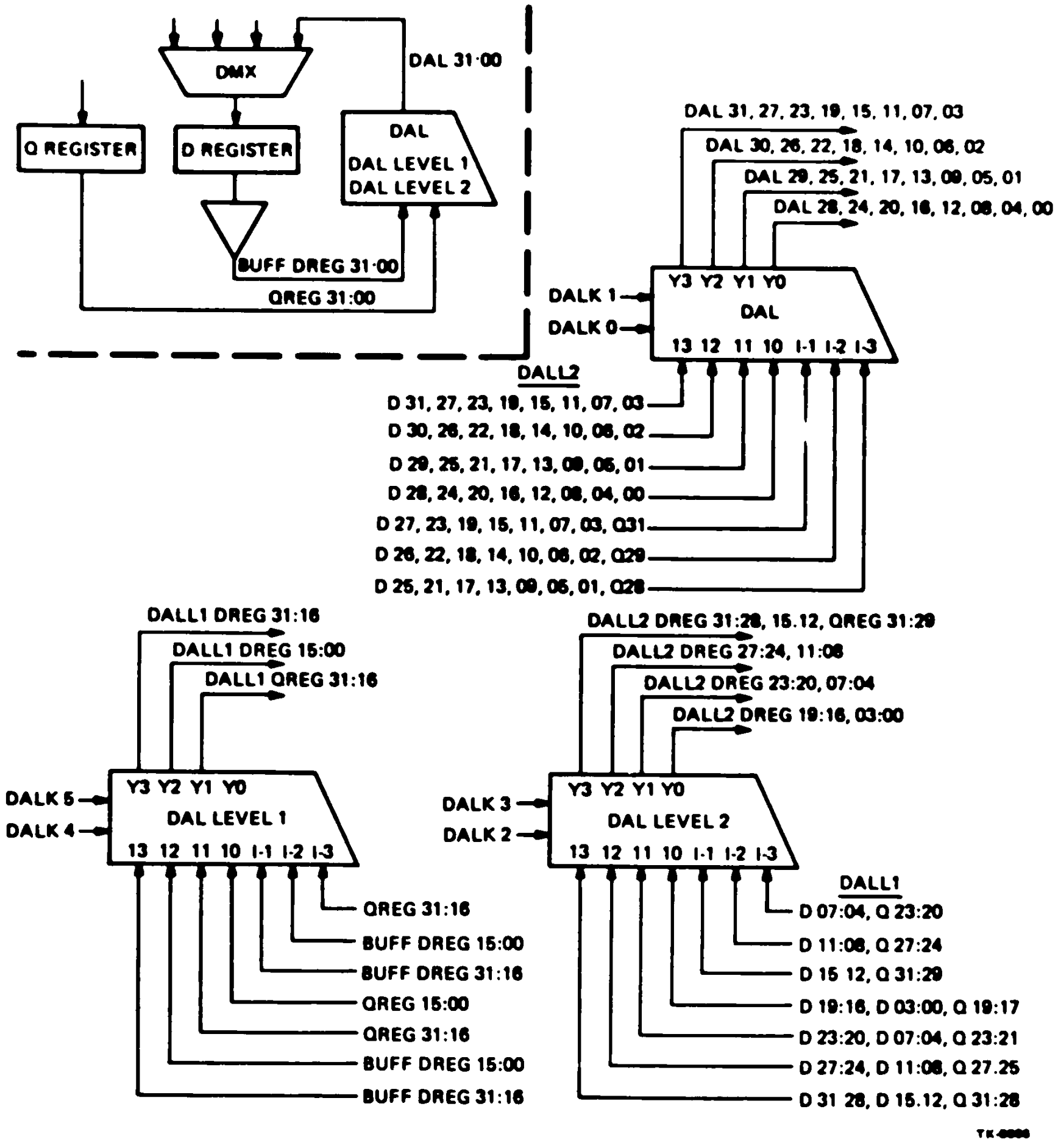


Figure 3-55 Data Aligner Logic

Table 3-31 DAL Level 1 Outputs To DAL Level 2

DALK (5:4)	DREG (31:16)	DREG (15:00)	QREG (31:16)	Level 1 Function
0 0	D(31:16)	D(15:00)	Q(31:16)	Shift by 0
0 1	D(15:00)	Q(31:16)	Q(15:00)	Left shift by 16
1 0	Q(31:16)	Q(15:00)	D(31.16)	Left shift by 32 (effective right shift by 3)
1 1	Q(15:00)	D(31 16)	Q(15:00)	Left shift by 48 (effective right shift by 1)

Level 1 performs a left shift of 0, 16, 32, or 48 bits as shown in Table 3-31 A left shift of 32 bits is, effectively, a right shift of 32 bits while a left shift of 48 bits is, effectively, a right shift of 16 bits Level 2 then produces a left shift of 0, 4, 8, or 12 bits as shown in Table 3-32 and the final DAL level produces a left shift of 0, 1, 2, or 3 bits as shown in Table 3-33

The shift value selected by the DAL logic is determined by the shift count register (SC (9,04:00)) or by a number (NORM SHF VAL) generated to normalize the fraction field in floating-point instructions The LDK field of the microinstruction selects the source of the DAL inputs as shown in Figure 3-51

3.4.2.3 Memory Data (MD) Bus Interface - The MD bus lines handle data transfers between the data section and memory Memory data is read or written on longword boundaries that are aligned to byte 0 Data section references to memory locations are made on byte boundaries, however, requiring that the data be properly byte aligned on the transfer This alignment is provided by the memory data aligner (MDAL) and byte aligner (BAL), and the BUS MD parity aligner and BUS MD mask generator shown in Figure 3-56

Memory Data Aligner (MDAL) - On a memory read, the D register is loaded with MD bus data from the MDAL through the D register multiplexer (DMX) The MDAL rotates the MD bus data and asserts it to the DMX according to the byte address of the memory reference

The shifting for all data types is controlled by the value of the two lowest virtual address bits (VA(01:00)) of the memory reference These bits identify the location of the first valid byte in the longword, rotating the memory data to the MDAL outputs as shown in Table 3-34

Table 3-32 DAL Level 2 Outputs to DAL

DALK (03:02) Value	DREG (31:28)	DREG (27:24)	DREG (23:20)	DREG (19:16)	DREG (15:12)	DREG (11:08)	DREG (07:04)	DREG (03:00)	QREG (31:29)	Level 2 Function
0 0	D(31 28)	D(27 24)	D(23 20)	D(19:16)	D(15 12)	D(11:08)	D(07:04)	D(03:00)	Q(31 29)	Shift by 0
0 1	D(27 24)	D(23 20)	D(19:16)	D(15 12)	D(11:08)	D(07:04)	D(03:00)	Q(31 28)	Q(27 25)	Left shift by 4
1 0	D(23 20)	D(19 16)	D(15 12)	D(11:08)	D(07:04)	D(03:00)	Q(31 28)	Q(27 24)	Q(23 21)	Left shift by 8
1 1	D(19:16)	D(15 12)	D(11:08)	D(07:04)	D(03:00)	Q(31 28)	Q(27 24)	Q(23 20)	Q(19 17)	Left shift by 12

Table 3-33 DAL Outputs To DMX

DALK (01:00) Value	DAL (31:28)	DAL (27:24)	DAL (23:20)	DAL (19:16)	DAL (15:12)	DAL (11:08)	DAL (07:04)	DAL (03)	DAL (02)	DAL (01)	DAL (00)	DAL Function
0 0	D(31.28)	D(27 24)	D(23 20)	D(19:16)	D(15 12)	D(11:08)	D(07:04)	D(3)	D(2)	D(1)	D(0)	Shift by 0
0 1	D(30:27)	D(26 23)	D(22 19)	D(18:15)	D(14 11)	D(10:07)	D(06:03)	D(2)	D(1)	D(0)	Q(31)	Left shift by 1
1 0	D(29.26)	D(25 22)	D(21 18)	D(17:14)	D(13 10)	D(09:06)	D(05:02)	D(1)	D(0)	Q(31)	Q(30)	Left shift by 2
1 1	D(28 25)	D(24.21)	D(20:17)	D(16:13)	D(12:09)	D(08:05)	D(04:01)	D(0)	Q(31)	Q(30)	Q(29)	Left shift by 3

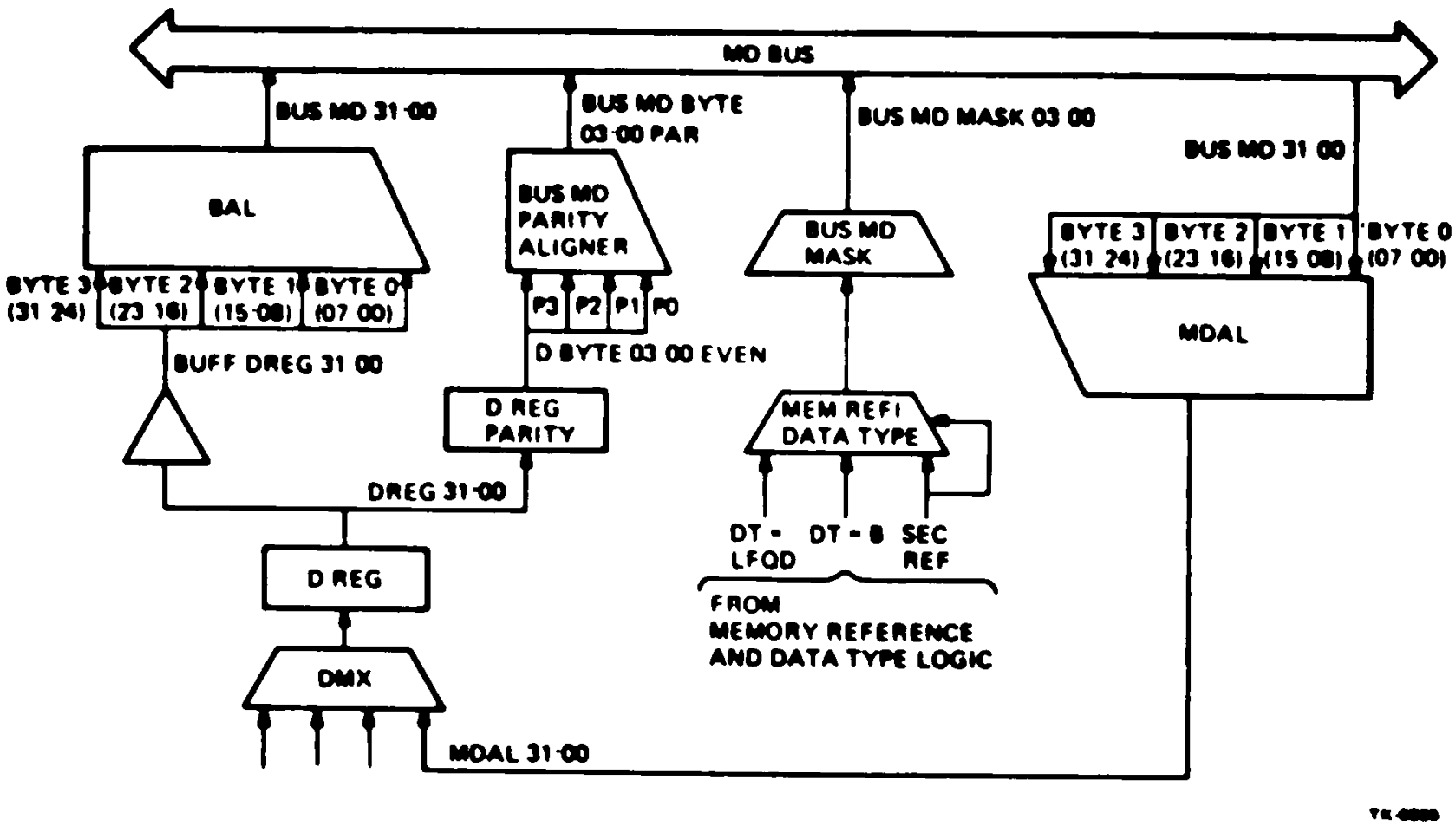


Figure 3-56 Memory Data (MD) Bus Interface

Table 3-34 VA(01:00) Control of the MDAL

VA Bits (01)	VA Bits (00)	Memory Data Aligner (MDAL) Outputs Byte 3	Byte 2	Byte 1	Byte 0
0	0	Byte 3	Byte 2	Byte 1	Byte 0
0	1	Byte 0	Byte 3	Byte 2	Byte 1
1	0	Byte 1	Byte 0	Byte 3	Byte 2
1	1	Byte 2	Byte 1	Byte 0	Byte 3

If the instruction specifies a byte data type, only byte 0 of the D register contains useful data after the read. (The RAMX or RBMX then zero or sign-extends the D register before it is transferred to the arithmetic section.) If a word data type is specified, bytes 0 and 1 then contain valid data.

A second memory reference is required if a longword data type is specified and VA (01:00) are not equal to 0. Because of the longword data type, the two low bits of the memory reference address (VA (01:00) have the same value on the second fetch. The MDAL again rotates the data so that the subsequent valid memory bytes are loaded into the correct bytes of the D register. The D register write enable logic prevents writing the bytes that already hold valid data, filling the D register with four bytes of valid data.

The D register write enable logic controls the loading of the D register on a per-byte basis. All bytes are written on the first memory reference with the valid bytes indicated by the byte mask. Enabling the D register bytes and determining whether a second memory reference is necessary depends on the reference data type and the two low order bits of the virtual byte address (VA (01:00)) as shown in Table 3-35 (a "1" means enabled and a "0" means disabled).

Table 3-35 VA⟨01:00⟩ Control of D Register Bytes

Data Type	VA Bits ⟨01⟩	VA Bits ⟨00⟩	Second Reference	D Register Write Enable Byte 3	Byte 2	Byte 1	Byte 0
Byte	0	0	No	1	1	1	1
	0	1	No	1	1	1	1
	1	0	No	1	1	1	1
	1	1	No	1	1	1	1
Word	0	0	No	1	1	1	1
	0	1	No	1	1	1	1
	1	0	No	1	1	1	1
	1	1	Yes	0	0	1	0
Long-word	0	0	No	1	1	1	1
	0	1	Yes	1	0	0	0
	1	0	Yes	1	1	0	0
	1	1	Yes	1	1	1	0

Byte Aligner (BAL) - On a memory write, D register data is asserted to the MD bus lines from the byte aligner (BAL) BAL functions are similar to the MDAL, but rotation takes place in the reverse direction. The BAL rotates the D register data according to the byte address of the memory reference and asserts the entire longword on the MD bus (The bytes that are valid for writing to memory are specified by the byte mask)

The shifting for all data types is controlled by the value of the two lowest virtual address bits (VA ⟨01 00⟩) of the memory reference These bits identify the location of the first valid byte in the longword, rotating the D register data to the MDAL outputs as shown in Table 3-36

Table 3-36 VA ⟨01:00⟩ Control of the BAL

VA Bits ⟨01⟩	VA Bits ⟨00⟩	Byte Aligner (BAL) Outputs to the MD Bus Byte 3	Byte 2	Byte 1	Byte 0
0	0	Byte 3	Byte 2	Byte 1	Byte 0
0	1	Byte 2	Byte 1	Byte 0	Byte 3
1	0	Byte 1	Byte 0	Byte 3	Byte 2
1	1	Byte 0	Byte 3	Byte 2	Byte 1

BUS MD Parity Aligner - The D register parity generator provides one odd parity bit for each byte of data and transmits them with the D register data on the ID bus or MD bus When parity is transmitted on the MD bus, each parity bit is aligned with its associated data byte by the BUS MD parity aligner which provides the rotation of the parity bit pattern (from VA ⟨01:00⟩) as shown in Table 3-37

Table 3-37 VA ⟨01:00⟩ Control of D Register Byte Parity

VA Bits ⟨01⟩	VA Bits ⟨00⟩	D Register Parity Bit Outputs P3	P2	P1	P0
0	0	P3	P2	P1	P0
0	1	P2	P1	P0	P3
1	0	P1	P0	P3	P2
1	1	P0	P3	P2	P1

Byte Mask Generator - On a memory read or write, a byte mask field is generated and transmitted on the MD bus as shown in Table 3-38 On a read, the mask field specifies which bytes of a longword should be checked by the memory controller for data integrity. On a write, the mask field specifies which bytes of the longword are valid data for writing to memory as follows. If a second memory reference is necessary because of the starting memory address and data type, a second mask field is generated.

Table 3-38 VA ⟨01:00⟩ Control of the Byte Mask

Data Type	VA Bits ⟨01⟩	VA Bits ⟨00⟩	Second Reference	Mask No.	BUS MD Byte Mask Bits B3	B2	B1	B0
Byte	0	0	No	1	0	0	0	1
	0	1	No	1	0	0	1	0
	1	0	No	1	0	1	0	0
	1	1	No	1	1	0	0	0
Word	0	0	No	1	0	0	1	1
	0	1	No	1	0	1	1	0
	1	0	No	1	1	1	0	0
	1	1	Yes	1	1	0	0	0
				2	0	0	0	1
Long-word	0	0	No	1	1	1	1	1
	0	1	Yes	1	1	1	1	0
				2	0	0	0	1
	1	0	Yes	1	1	1	0	0
				2	0	0	1	1
	1	1	Yes	1	1	0	0	0
				2	0	1	1	1

3.4.3 Arithmetic Section

The arithmetic section performs arithmetic and logic operations and constant generation. It contains the general processor and temporary storage registers. For arithmetic processing, data type context is defined by the UDT field of the CPU microinstruction as shown in Table 3-39.

Table 3-39 UDT Field Context Control

UDT (01:00)	Context
0	Longword (Longword, Quadword, Floating, or Double)
1	Word
2	Byte
3	Instruction dependent (context determined by the instruction decode logic)

3.4.3.1 Arithmetic/Logic Unit (ALU) – The ALU is the main processing unit of the arithmetic section. It performs logical functions or arithmetic operations (with carry look ahead) on longword data types. Byte or word data types are zero or sign-extended to 32 bits before being processed. The ALU processes information from registers in the data and address sections and is the main transfer path for address and data information between other sections of the CPU data path logic.

Register contents from the address, data, and exponent sections are asserted to the ALU through the A and/or B input multiplexers (AMX and BMX). The ALU operation select inputs are controlled by the UALU field of the CPU microword as shown in Table 3-40. If the UALU field is equal to 3 (instruction dependent mode), the function is determined by the instruction decode logic on the data path control (DAP) module. The instruction dependent ROMs are then addressed by the instruction opcode from the IRC module, selecting all ALU logic and arithmetic functions as shown in Table 3-41.

When the UALU field equals 1 or 6, the register log (RLOG) stack is updated with the general register address (RA), the lower four bits of the constant multiplexer (KMX), and a bit that determines if an add or subtract is requested The RLOG stack contains 16 locations that keep track of changes made to the scratchpad register set.

3.4.3.2 ALU A Input Multiplexer (AMX) – The AMX transfers data section information or scratchpad register set A (RA) contents to the A input of the ALU as shown in Figure 3-57. Data used during instruction execution comes from the Q or D register in the data section or from the scratchpad register set A and is asserted to the ALU from the AMX.

For scratchpad register set A data, the AMX selects the latch A (LA (31:00)) output of the register set after the contents of the register location are stored in the latch. For Q or D register data, the AMX selects the register A multiplexer (RAMX (31:00)) in the data section for the ALU input. (The Q or D register is selected by the CPU microword as described in Section 3.4.2.1) If the selected register contents are less than 32 bits, they are sign- or zero-extended to 32 bits by the RAMX. The AMX is controlled by the UAMX field of the CPU microword as shown in Table 3-42.

The data type and required sign or zero extension of the RAMX is then determined by the value of the UDT field as shown in Table 3-43. The zero extension of a longword format results in all zeros at the AMX output.

Table 3-40 UALU Field ALU Select Functions

UALU (03:00)	ALU Select (S3:S0)	ALU Mode Input*	ALU Carry Input	ALU Function
0	6	0	1	A minus B
1	6	0	1	A minus B (RLOG)
2	6	0	1	A minus B minus 1
3	-	-	-	Instruction dependent
4	9	0	1	A plus B plus 1
5	9	0	0	A plus B
6	9	0	0	A plus B (RLOG)
7	D	1	1	A OR $\overline{B}$
8	6	1	1	A XOR B
9	7	1	1	A AND $\overline{B}$
A	0	1	1	$\overline{A}$
B	9	0	Conditional on PSL C-bit	A plus B plus C
C	E	1	1	A OR B
D	B	1	1	A AND B
E	A	1	1	B
F	F	1	1	A

* 1 means logic mode (FLn__), 0 means arithmetic mode (FLn).

Table 3-41 ALU Functions in Instruction-Dependent Mode

ALU Select (S3:S0)	Logic Functions (M = 1)	Arithmetic Functions (M = 0) Cn = 0 (no carry)	Cn = 1 (with carry)
0	$\overline{A}$	A	A plus 1
1	$\overline{A}$ OR $\overline{B}$	A OR B	(A OR B) plus 1
2	$\overline{A}$ AND B	A OR $\overline{B}$	(A OR $\overline{B}$) plus 1
3	0	Minus 1 (2's complement)	0
4	$\overline{A}$ AND $\overline{B}$	A plus (A AND $\overline{B}$)	A plus (A AND $\overline{B}$) plus 1
5	$\overline{B}$	(A OR B) plus (A AND B)	(A OR B) plus (A AND B) plus 1
6	A XOR B	A minus B minus 1	A minus B
7	A AND $\overline{B}$	(A AND $\overline{B}$) minus 1	A AND $\overline{B}$
8	$\overline{A}$ OR B	A plus (A AND B)	A plus (A AND B) plus 1
9	$\overline{A}$ XOR $\overline{B}$	A plus B	A plus B plus 1
A	B	(A OR $\overline{B}$) plus (A AND B)	(A OR $\overline{B}$) plus (A AND B) plus 1
B	A AND B	(A AND B) minus 1	A AND B
C	1	A plus A*	A plus A plus 1
D	A OR $\overline{B}$	(A OR B) plus A	(A OR B) plus A plus 1
E	A OR B	(A OR $\overline{B}$) plus A	(A OR $\overline{B}$) plus A plus 1
F	A	A minus 1	A

* Each bit shifts to the next more significant position.

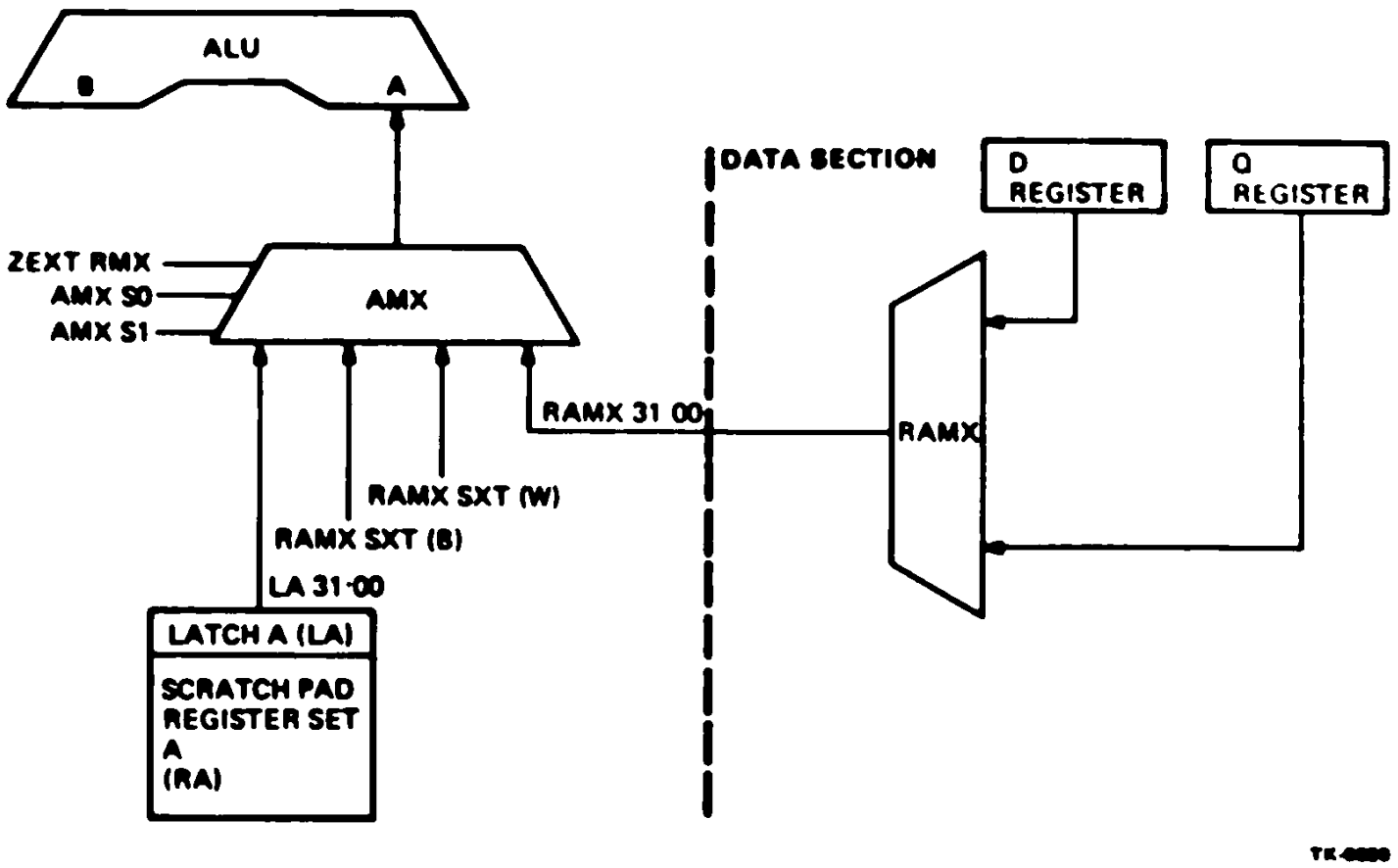


Figure 3-57 ALU A Input Multiplexer (AMX)

Table 3-42 UAMX Field Control of the AMX

UAMX ⟨01:00⟩	Data Selected
0	LA ⟨31:00⟩
1	RAMX ⟨31:00⟩
2	RAMX SXT (Sign extension determined by UDT field)
3	RAMX 0XT (Zero extension determined by UDT field)

Table 3-43 UDT Field Control of the RAMX

UDT ⟨01:00⟩	Data Type
0	Longword: SXT(L) or all zeros
1	Word: SXT(W) or 0XT(W)
2	Byte: SXT(B) or 0XT(B)
3	Instruction dependent

When the UDT field equals 3, the data type is determined by the instruction decode logic which provides information for instruction execution and operand specifier evaluation Instructions requesting quadword, floating, or double-floating context result in the longword data type. The UAMX field of the CPU microword controls the data format of the AMX outputs as shown in Figure 3-58.

3.4.3.3 ALU B Input Multiplexer (BMX) - The BMX transfers data or address section information or selects information from the exponent, sign, or arithmetic section registers for the B input of the ALU as shown in Figure 3-59 These inputs are used separately or in combination for the following operations.

Address Generation - The program counter (PC) updating takes place by routing the PC from the address section to the ALU through the BMX In displacement addressing modes using the PC, for example, the new PC value is calculated by selecting the current PC value from the BMX and the sign extended displacement value from the AMX

Manipulation of Stored Data In certain operations, the BMX is used for the same purposes as the AMX Information stored in the data section (D or Q registers) and in scratchpad register set B are input to the ALU through the BMX The contents of register set B are an exact copy of the register set A contents.

Having the same information available at both ALU inputs allows fast access to the source and destination registers during the execution of register to register instructions This also allows instructions using the B MINUS A function to be executed without swapping the operands between the ALU inputs (The B MINUS A function is not provided by the ALU) Other data required for the execution of instructions is also stored in scratchpad register set C.

Instruction Restart The RLOG and PCSV inputs to the BMX are used to restart an instruction If a fault occurs, the entire 32-bit PC can be reconstructed from the contents of the PC save (PCSV) register and the general registers can be restored from the RLOG stack (as described in Section 3.4.3.8).

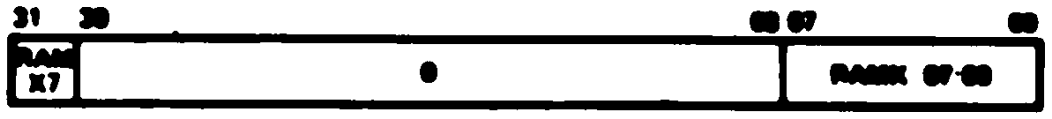
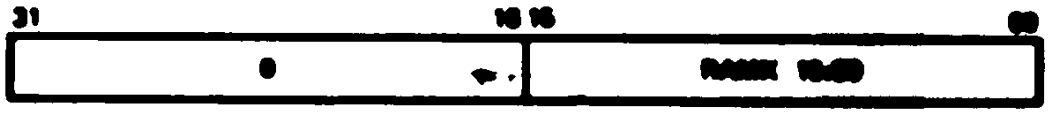
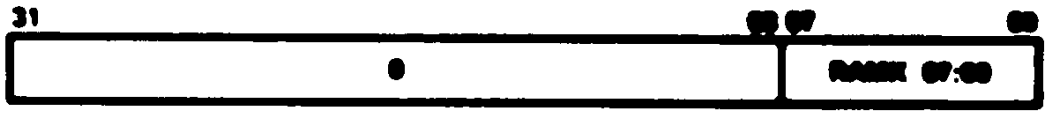
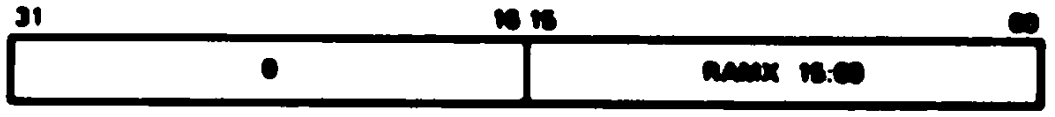
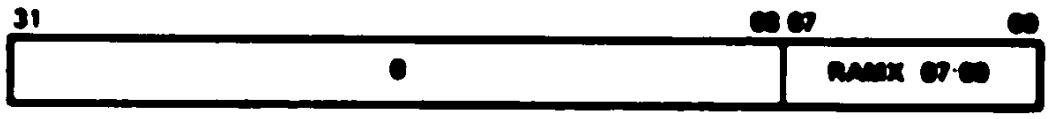
The PCSV register holds the lower eight bits of the PC at the beginning of an instruction. The RLOG stack contains 16 locations that store a record of changes made to the scratchpad register set The RLOG and PCSV inputs are selected when the UBMX microword field is equal to 0 and the signal READ RLOG is asserted

Mask and Constant Generation - The MASK input to the BMX routes the outputs of the bit mask generator to the ALU for logic operations (as described in Section 3 4 3 6) The constant multiplexer (KMX) input supplies constants required for the execution of instructions and evaluation of operand specifiers (as described in Section 3 4 3 5).

Assembly of Floating Point Data Formats - During the execution of floating-point instructions, inputs from the data, exponent, and control section are assembled by the BMX for input to the ALU, forming a packed floating-point data type

For the packed floating-point data type format, the fraction position is taken from the DREG and the sign (SD) from the control logic The exponent is taken from the EALU as shown in Figure 3-59

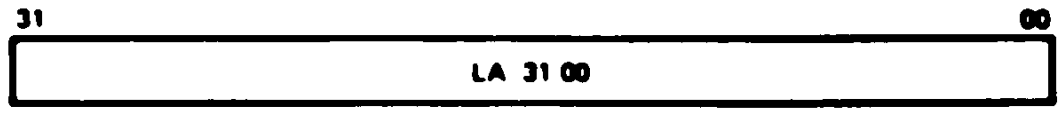
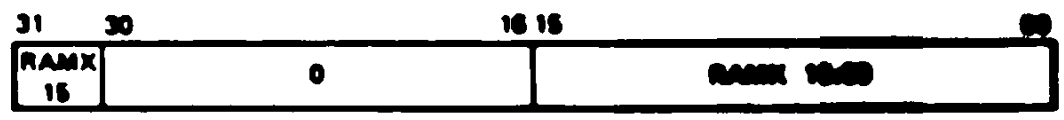
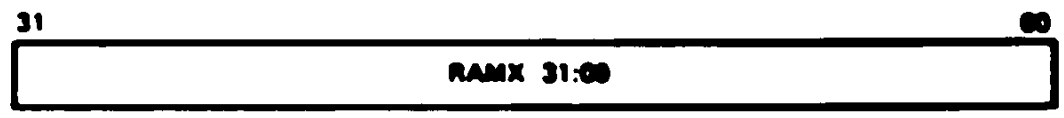
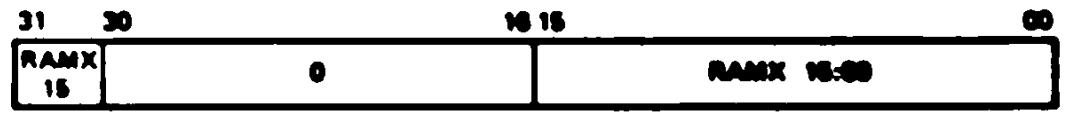
The SD bit contains the sign of the destination fraction in floating-point operations It is loaded and controlled by the USGN field of the CPU microword during execution of a floating point instruction The source sign (SS) and destination sign (SD) are selected by the USGN field of the CPU microword as shown in Table 3-44

UAMX 1	UAMX 0	UBT 1	UBT 0	UBT DISP DATA TYPE L	W	B	AMX DATA FORMAT	SIGN OR ZERO EXTENSION
H	L	H	H	L	L	H		SXT BYTE
H	H	L	L	X	X	X		OXT LONG
H	H	L	H	X	X	X		OXT WORD
H	H	H	L	X	X	X		OXT BYTE
H	H	H	H	H	L	L		OXT LONG
H	H	H	H	L	H	L		OXT WORD
H	H	H	H	L	L	H		OXT BYTE

X=IRRELEVANT HAVE NO EFFECT

MEV84 1366

Figure 3-58 A Input Multiplexer (AMX) Control
(Sheet 1 of 2)

UAMIX 1 0	UDT 1 0	INST DEP DATA TYPE L W B	AMX DATA FORMAT	SIGN OR ZERO EXTENSION
L L	X X	X X X		
L H	X X	X X X		
H L	L L	X X X		SXT LONG
H L	L H	X X X		SXT WORD
H L	H L	X X X		SXT BYTE
H L	H H	H L L		
H L	H H	L H L		SXT WORD

X=IRRELEVANT

MKV84 1370

Figure 3-58 A Input Multiplexer (AMX) Control
(Sheet 2 of 2)

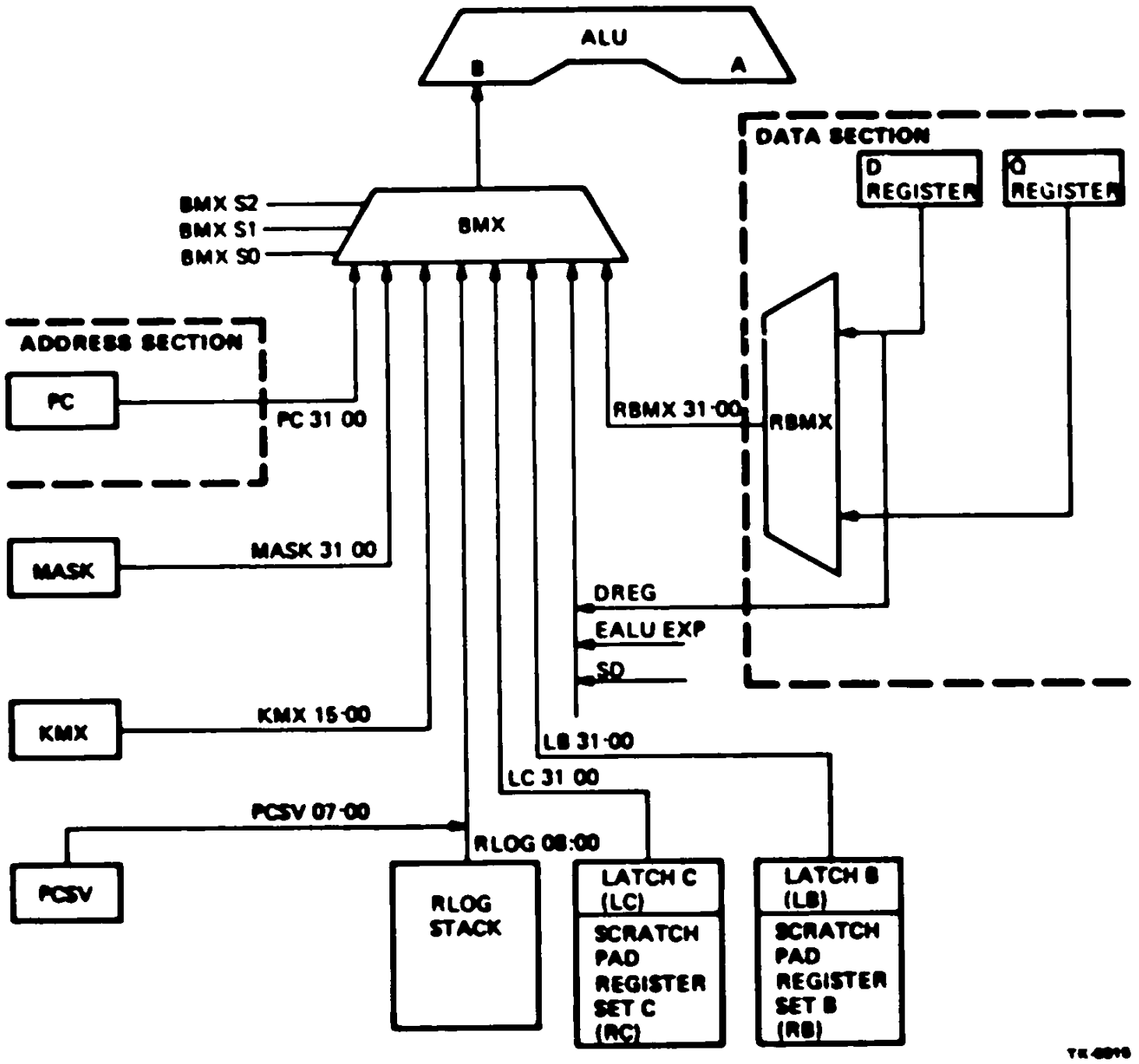


Figure 3-59 ALU B Input Multiplexer (BMX)

Table 3-44 USGN Field Control of the Source and Destination Sign

USGN (02:00)	Source Sign (SS)	Destination Sign (SD)
0	SS	SD
1	ALU (15)	SD
2	SD	SD
3	SS	SD
4	SS	SS
5	ALU (15) XOR SS XOR IR (1)	ALU (15)
6	ALU (15) XOR SS	ALU (15)
7	0	0

Because of the time delay in routing floating-point data, both the FALL and ALL are selected for logic mode to allow the data to be available in the arithmetic section when it is required

BMX input selection is controlled by the UBMX field of the CPU microword with two enabling conditions R = PC and READ RLOG as shown in Table 3-45 The BMX data format for each selected input is shown in Figure 3-60

Table 3-45 UBMX Field Control of the BMX

UBMX (02:00)	RLOG READ	R=PC	BMX SELECT S2	S1	S0	BMX DATA
0	0	X*	0	0	0	MASK
0	1	X	0	0	1	RLOG and PCSV
1	0	0	0	1	1	LB
1	0	1	1	0	1	PC
2	0	X	0	1	0	Packed floating point
3	0	X	0	1	1	LB
4	0	X	1	0	0	LC
5	0	X	1	0	1	PC
6	0	X	1	1	0	KMX
7	0	X	1	1	1	RBMX

* X means "does not apply" Either state has no effect

3.4.3.4 ALU Shifter (SHF) - The ALU shifter logic shown in Figure 3-61 selects the data source for the scratchpad register sets and transfers information between the arithmetic and data sections It is used in the arithmetic section to multiply (left shift) or divide (right shift) the ALU outputs In index mode specifier evaluations, it is used as a multiplier to generate the correct index values for address calculations

Index mode addressing allows access to data arrays that are byte, word, longword, or quadword organized Access to an array requires the contents of an index register to be multiplied by the size of the array's primary operand in bytes (1 for byte, 2 for word, 4 for longword or floating operand, and 8 for quadword or double-floating operand) The SHF provides the required multiplication by shifting the data by the correct number of bits Multiplication by 1 requires no shift (L0), by 2 requires a left shift of 1 (L1), by 4 requires a left shift of 2 (L2); and by 8 requires a left shift of 3 (L3) The shift requirement, a function of the data type, is defined by the USHF field of the CPU microword or is controlled by the UDT field and the instruction decode logic

The SHF is also implemented in the execution of multiply, divide, compatibility mode rotate, and shift instructions These instructions force a left shift by 1 (L1), a right shift by 1 (R1), or a right shift by 2 (R2) Inputs to bit positions left empty by the shift are controlled by the shift input control (USI) field of the CPU microword as shown in Table 3-46

ALU shifter selection is determined by the value of the USHF field of the CPU microword as shown in Table 3-47 For USHF field values of 1, 2, or 4, the input to the vacated SHF bit positions is determined by the USI field For USHF field values 3 or 5, the vacated SHF positions are cleared

If the USHF field equals 3, the SHF output is controlled by the UDT field value as shown in Table 3-48

UBMX (HEX)	READ RLOG	R-PC	BMX DATA FORMAT
0	0	X	31 … 00: MASK 31 00
0	1	X	31 … 17 16 … 08 07 … 00: 00 | RLOG 08:00 | PCSV 07 00
1	0	0	31 … 00: LB 31:00
1	0	1	31 … 00: PC 31:00
2	0	X	31 … 16 15 14 … 07 06 … 00: D 23:08 | SD | EALU 07:00 | D 30 24
3	0	X	31 … 00: LB 31 00
4	0	X	31 … 00: LC 31:00
5	0	X	31 … 00: PC 31 00
6	0	X	31 … 16 15 … 00: 00 | KMX 15:00
7	0	X	31 … 00: RBMX 31:00

X=IRRELEVANT

MKV84-1364

Figure 3-60 BMX Data Formats on the ALU B Inputs

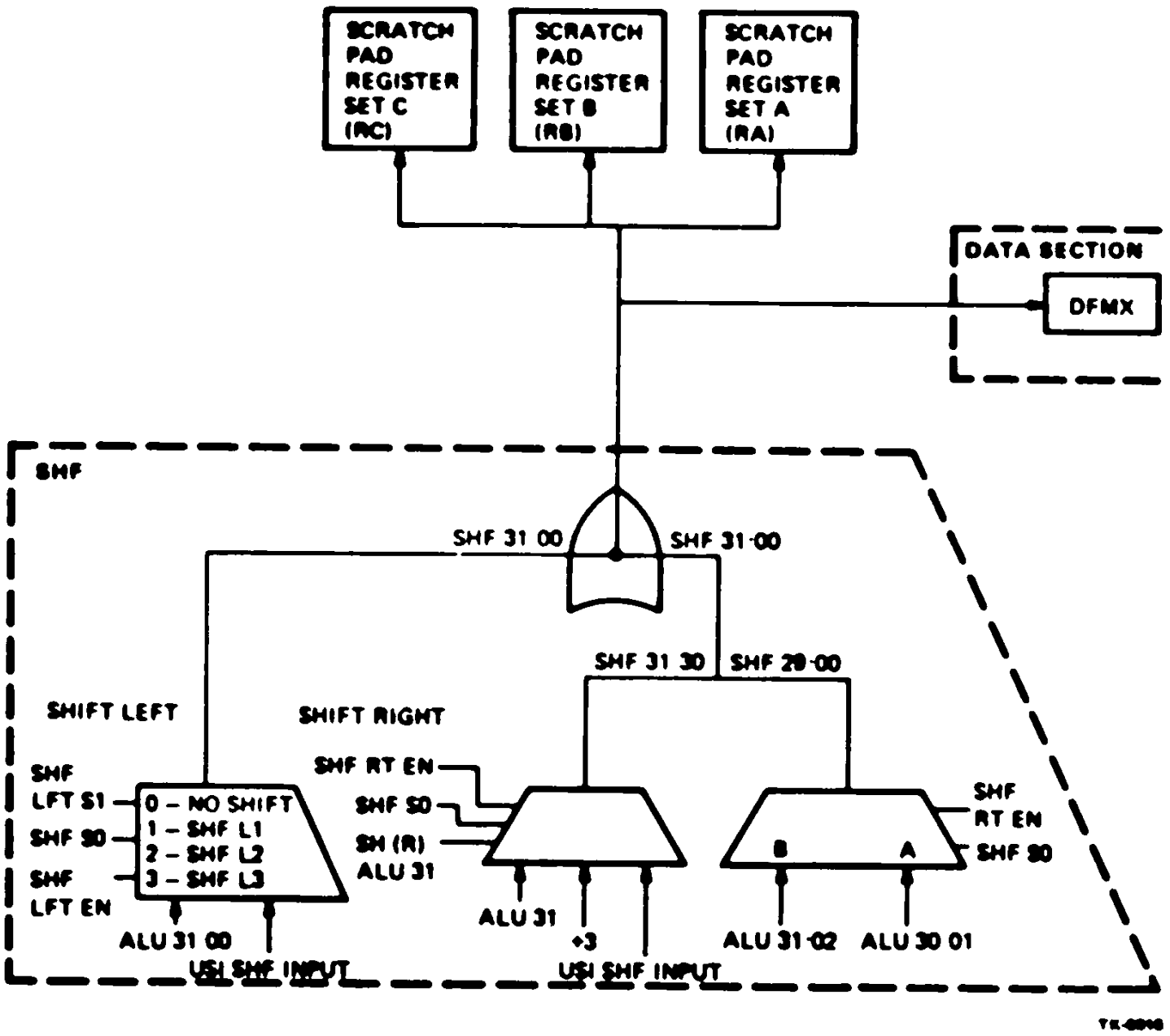


Figure 3-61 ALU Shifter (SHF) Logic

Table 3-46 USI Field Control of SHF Input

USI ⟨02:00⟩	Shift Input
0	PSL (N bit)
1	ALU (bit ⟨31⟩)
2	0
3	0
4	0
5	Q register (bit ⟨31⟩)
6	0
7	1

Table 3-47 USHF Field Control of SHF Output

USHF ⟨02:00⟩	ALU Shifter (SHF) Output
0	ALU with no shift (L0)
1	ALU left shifted by 1 (L1)
2	ALU right shifted by 1 (R1)
3	ALU shift determined by UDT field
4	ALU right shifted by 2 (R2)
5	ALU left shifted by 3 (L3)
6	Reserved
7	Reserved

Table 3-48 UDT Field Control of SHF Output (USHF Is Equal to 3)

UDT ⟨01:00⟩	ALU Shifter (SHF) Output by UDT Control
0	Longword: ALU left shifted by 2 (L2)
1	Word: ALU left shifted by 1 (L1)
2	Byte: ALU with no shift (L0)
3	Instruction dependent: any of the above, or quadword with ALU left shifted by 3 (L3)

If the UDT field equals 3, the SHF output is determined by the instruction decode logic (specifier 1 constant ⟨02:00⟩)

The USHF field of the CPU microcode controls the data format of the SHF outputs as shown in Figure 3-62 (USI indicates that the input bit value is determined by the USI field.)

3.4.3.5 Constant Multiplexer (KMX) - The KMX provides the arithmetic section with the constants required for certain arithmetic operations. It asserts them to the ALU through the BMX from either the fast constant multiplexer (FKMX) or the slow constant (SK) ROM as shown in Figure 3-63.

Fast Constant Multiplexer (FKMX) - The fast constant inputs are selected by the instruction decode logic, the UKMX field of the CPU microword, or the shift count (SC) register in the exponent section.

The instruction decode logic generates the specifier 1 and specifier 2 constants (SP1CON and SP2CON). In the native mode, SP1CON is the number 1, 2, 4, or 8, determined by the data type of the operand specifier being evaluated SP2CON is the number 0.

In compatibility mode, SP1CON is the number 1 or 2, determined by the data type and the register number of the instruction source register. SP2CON is the number 1 or 2, determined by the data type and the register number of the instruction destination register.

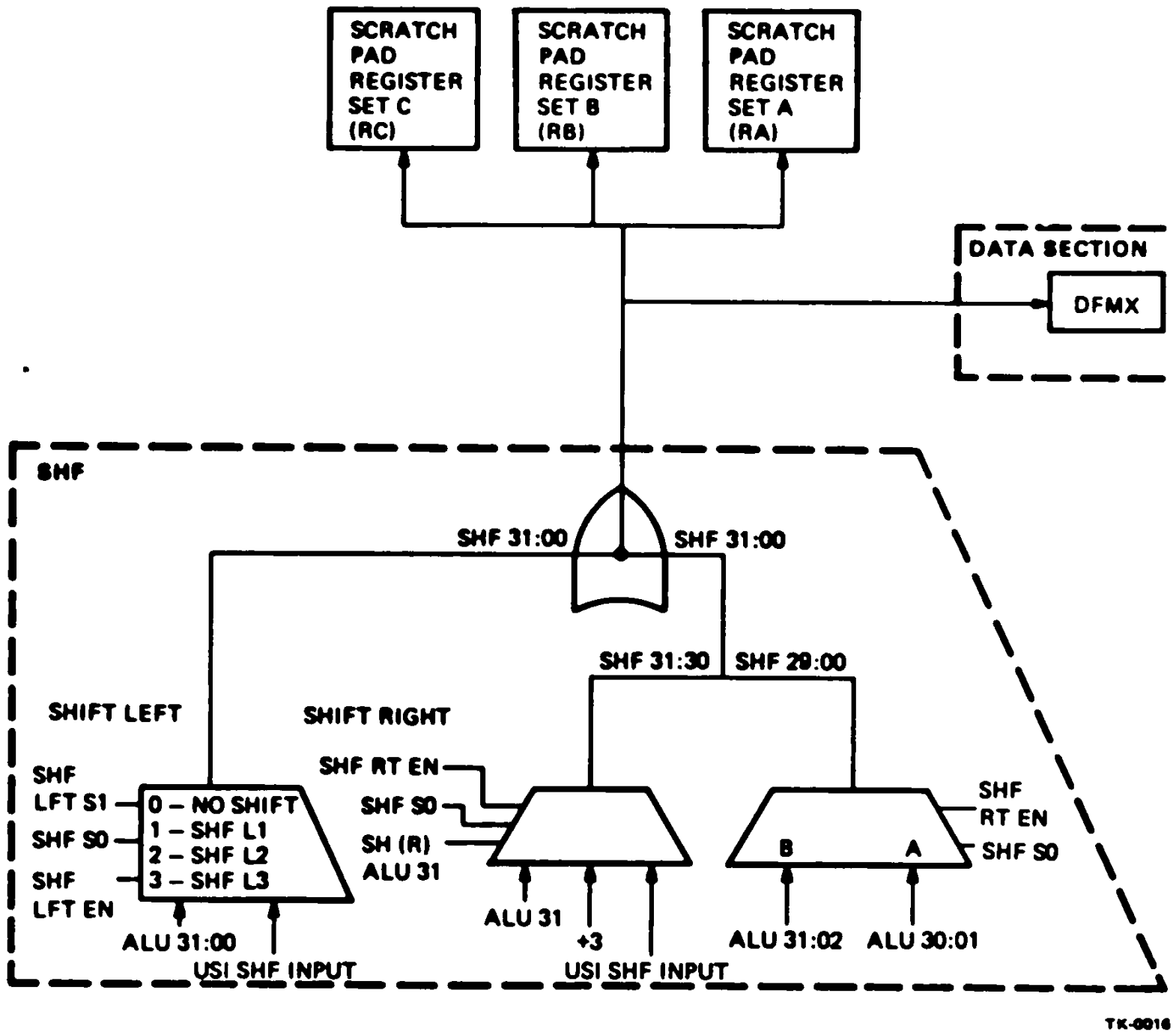


Figure 3-61 ALU Shifter (SHF) Logic

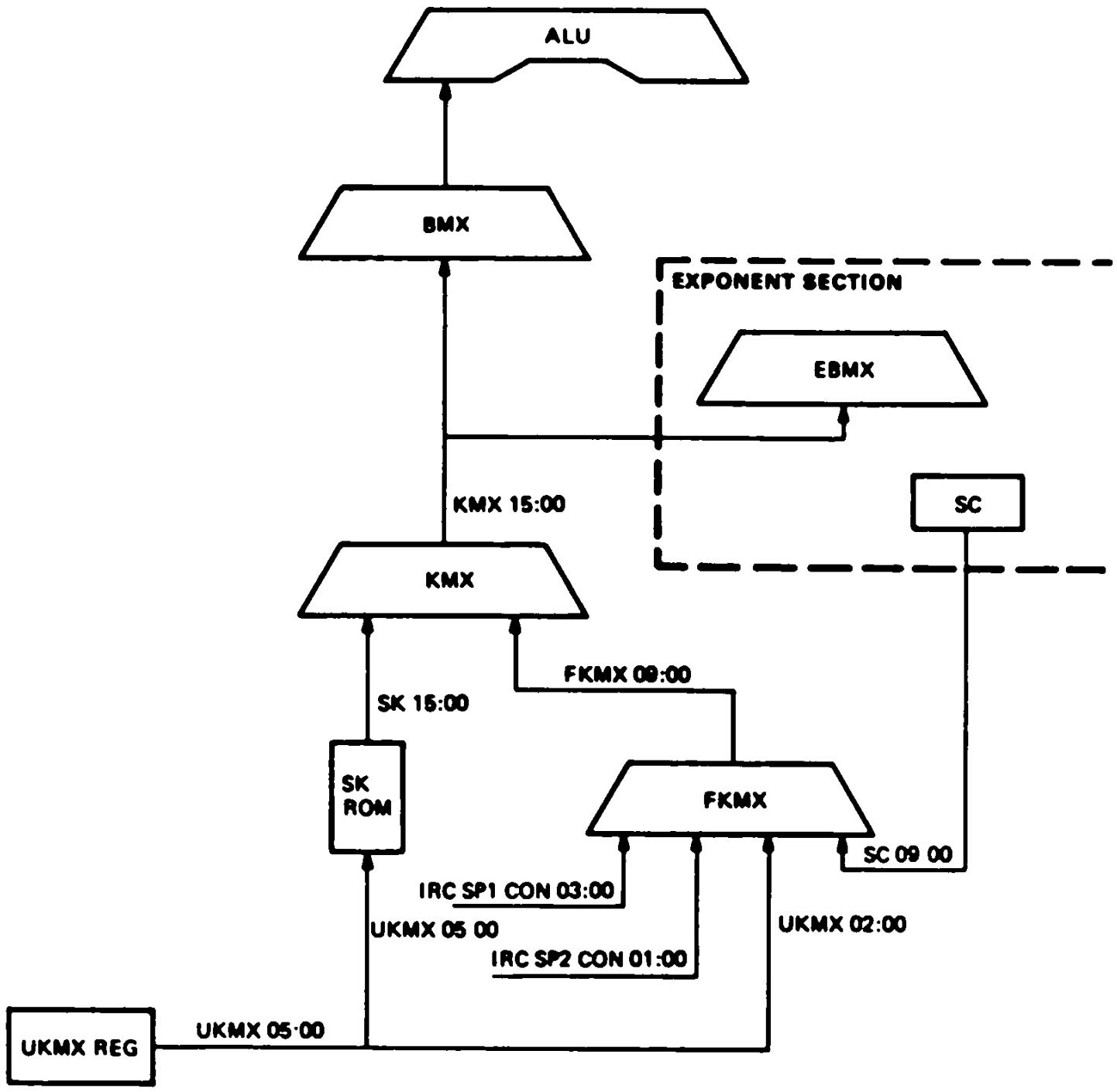


Figure 3-63 Constant Multiplexer (KMX)

Table 3-49 UKMX Field Constant Control

UKMX ⟨05:00⟩	KMX Output Constant
0	#8
1	#1
2	#2
3	#3
4	#4
5	SP1CON
6	SP2CON
7	SC ⟨09:00⟩
8 through 3F	SK ROM Constant (SK ⟨15:00⟩)

A constant of 1, 2, 3, 4, or 8 from the KMX may also be selected by the UKMX field of the CPU microword as shown in Table 3-49.

The shift count (SC) input to the FKMX provides a data path between the exponent and arithmetic sections. The shift count register is also used to store constants for arithmetic operations.

The fast constants are generally used to increment or decrement data and are implemented in the evaluation of the autoincrement and autodecrement modes.

Slow Constant (SK) ROM – The remainder of the microprogram constants are supplied by the SK ROM. These constants are used to execute instructions, isolate bits or bit fields, provide exponent biasing, and select shift constants.

3.4.3.6 Bit Mask (MASK) Generator – The MASK generator shown in Figure 3-64 generates a bit pattern within a 32-bit longword that is used to isolate bit fields through the ALU logic functions. This process occurs in the execution of bit field instructions and in the memory management process of translating virtual addresses to physical addresses from the translation buffer.

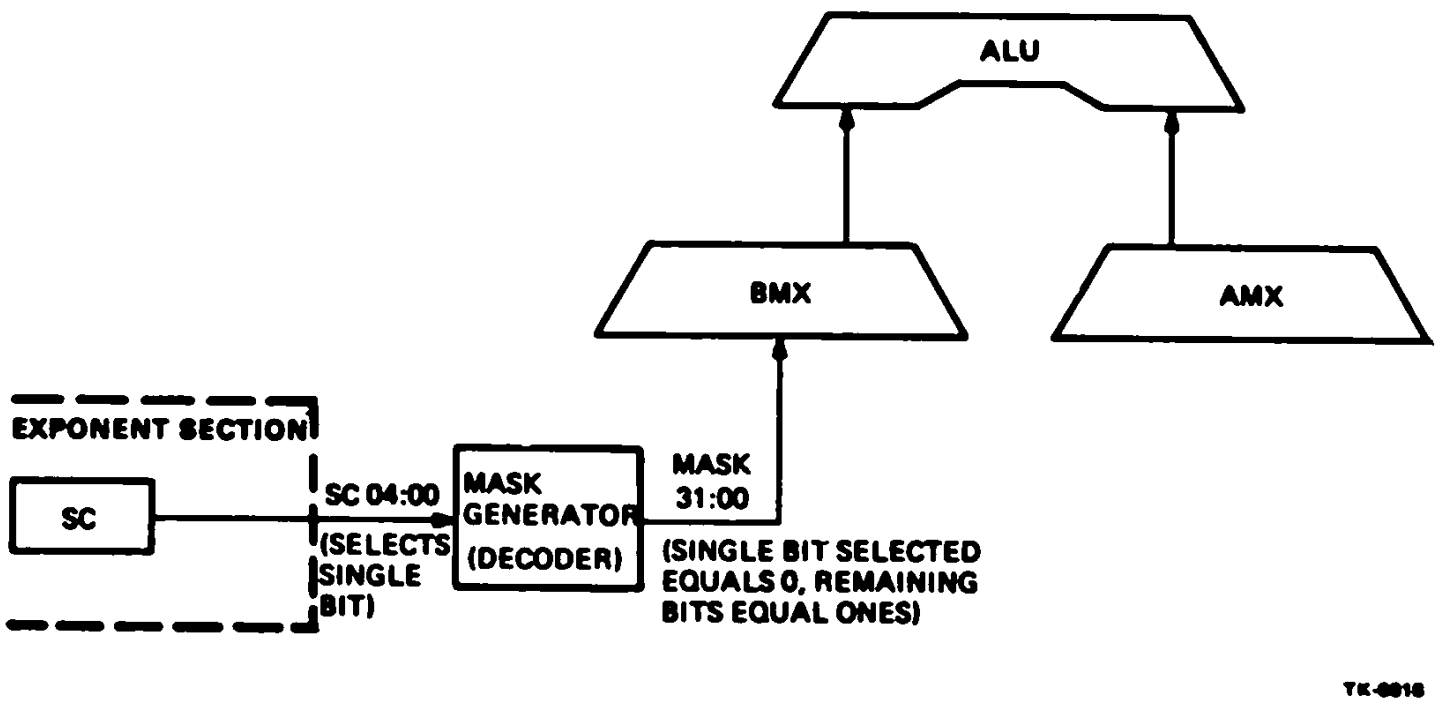


Figure 3-64 Bit Mask (MASK) Generator Logic

The shift count (SC) input selects the bit address of a single bit in a 32-bit longword that the MASK generator uses to produce a single 0 bit in a field of 1s.

The MASK output may be used to generate a bit pattern in or to extract a bit field from a 32-bit longword as shown in the following examples.

1. In the example shown in Figure 3-65, a pattern of all 1s is generated from bit position (05) and above. In this procedure, the AMX outputs equal 0, the SC is equal to 5 and the BMX asserts MASK.

   The MASK generator decodes the SC address and inserts a 0 in that position with the remaining output bits equal to 1s. The ALU operation A plus B plus 1 then produces the bit pattern.

2. In the example shown in Figure 3-66, the MASK generator output extracts a field of information from a 32-bit longword.

   If the field begins at bit P (equal to 9) and the field is of length S (equal to 8), two masks are generated to extract the field.

   The first mask is produced as shown in Figure 3-67. In this procedure, the AMX is equal to 0, the SC is equal to 9 and the BMX asserts MASK. The ALU operation is A plus B plus 1 and bit pattern A is stored in a temporary register for later use.

   The second mask is produced the same way but in the procedure, the AMX is equal to 0, the SC is equal to 17 (P plus S) and the BMX asserts MASK. The ALU operation is A plus B plus 1 and the result is bit pattern B as shown in Figure 3-68.

In the final procedure, the ALU logic function is then A ANDed with NOTB, resulting in the bit pattern shown in Figure 3-69. This resulting bit pattern can then be ANDed with a longword of data to extract an eight-bit field of information beginning at bit position (09).

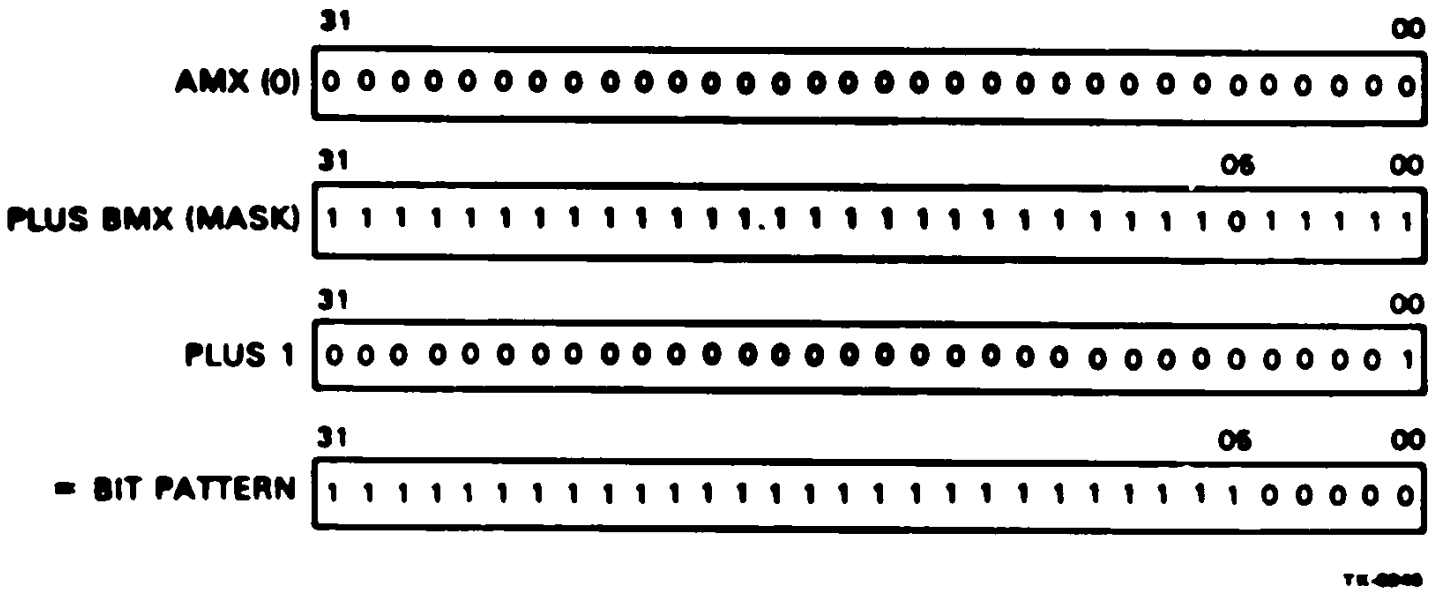


Figure 3-65 Example of Bit Pattern Generation

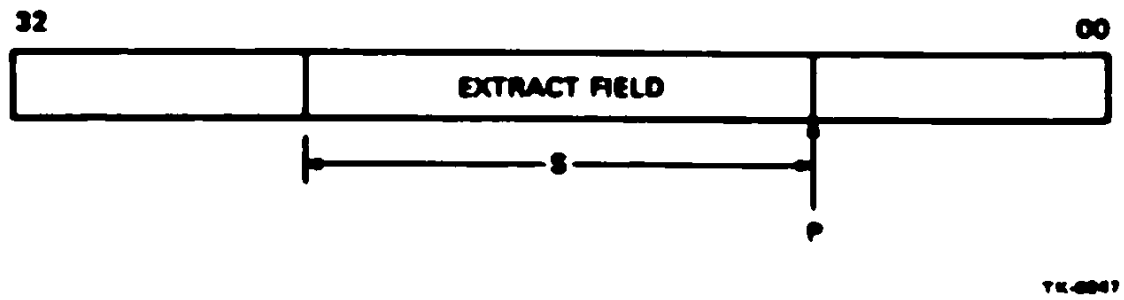


Figure 3-66 Bit Field to be Extracted

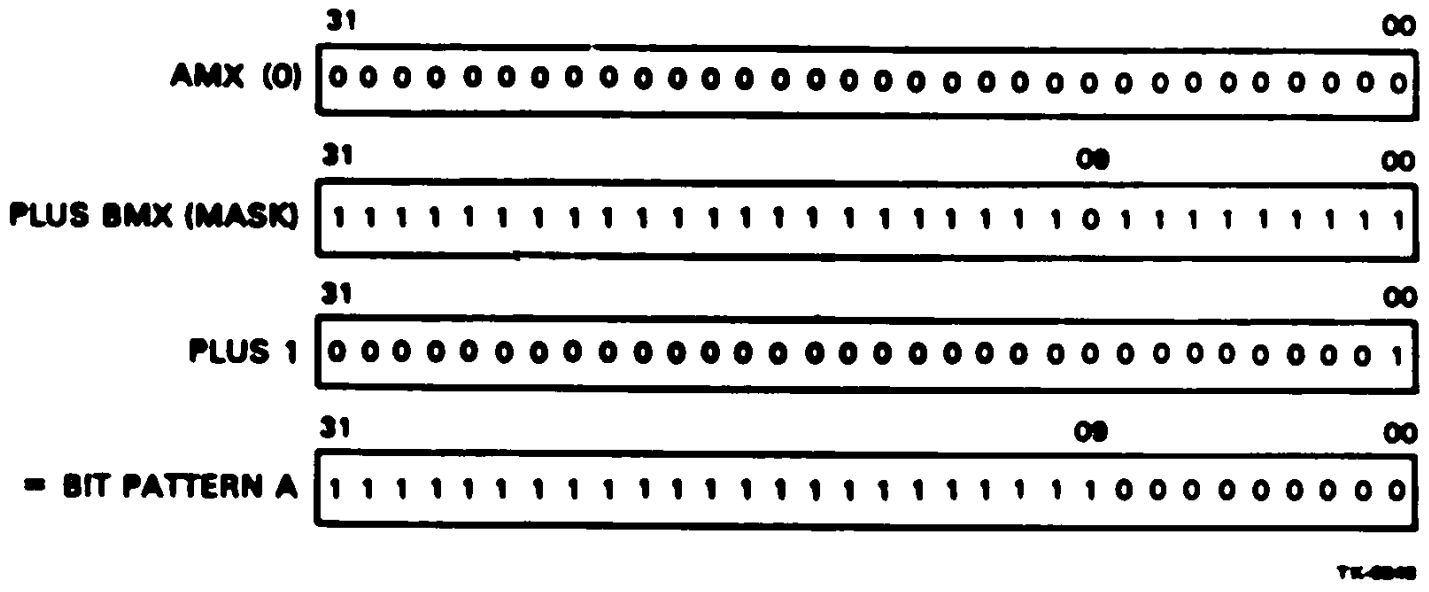


Figure 3-67 Generating Bit Pattern A

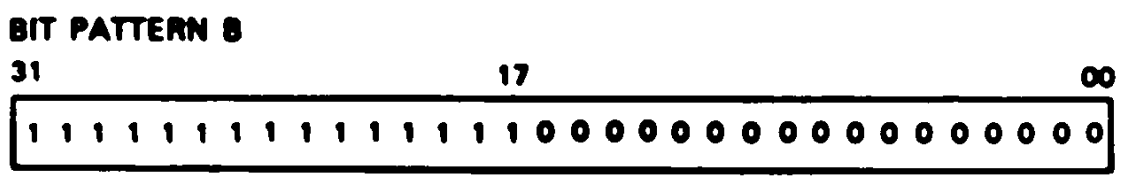


Figure 3-68 Resulting Bit Pattern B

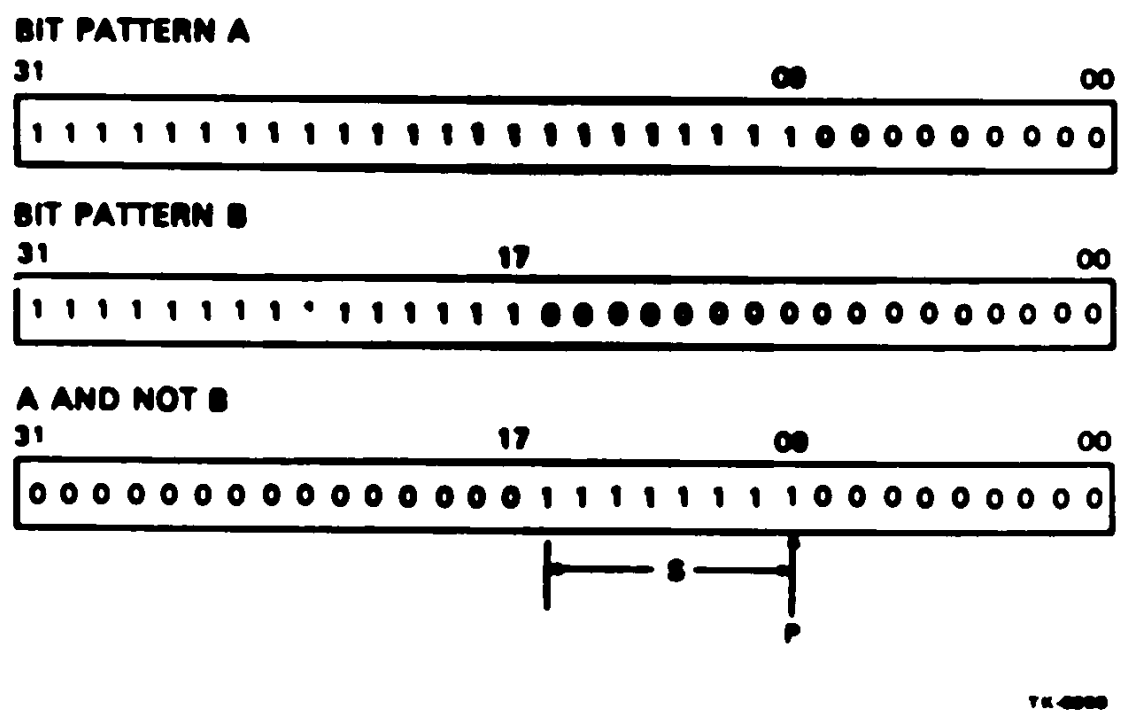


Figure 3-69 Resulting Bit Pattern of the Extract Field

3.4.3.7 Scratchpad Register Sets – Three 16 × 32-bit register sets are used for fast access temporary storage as shown in Figure 3-70.

Register Set C (RC) – RC is used as temporary storage for addresses and operands generated during the execution of the microprogram. On a register read, the RC latch (LC) holds the contents of the selected RC register for use by the arithmetic section.

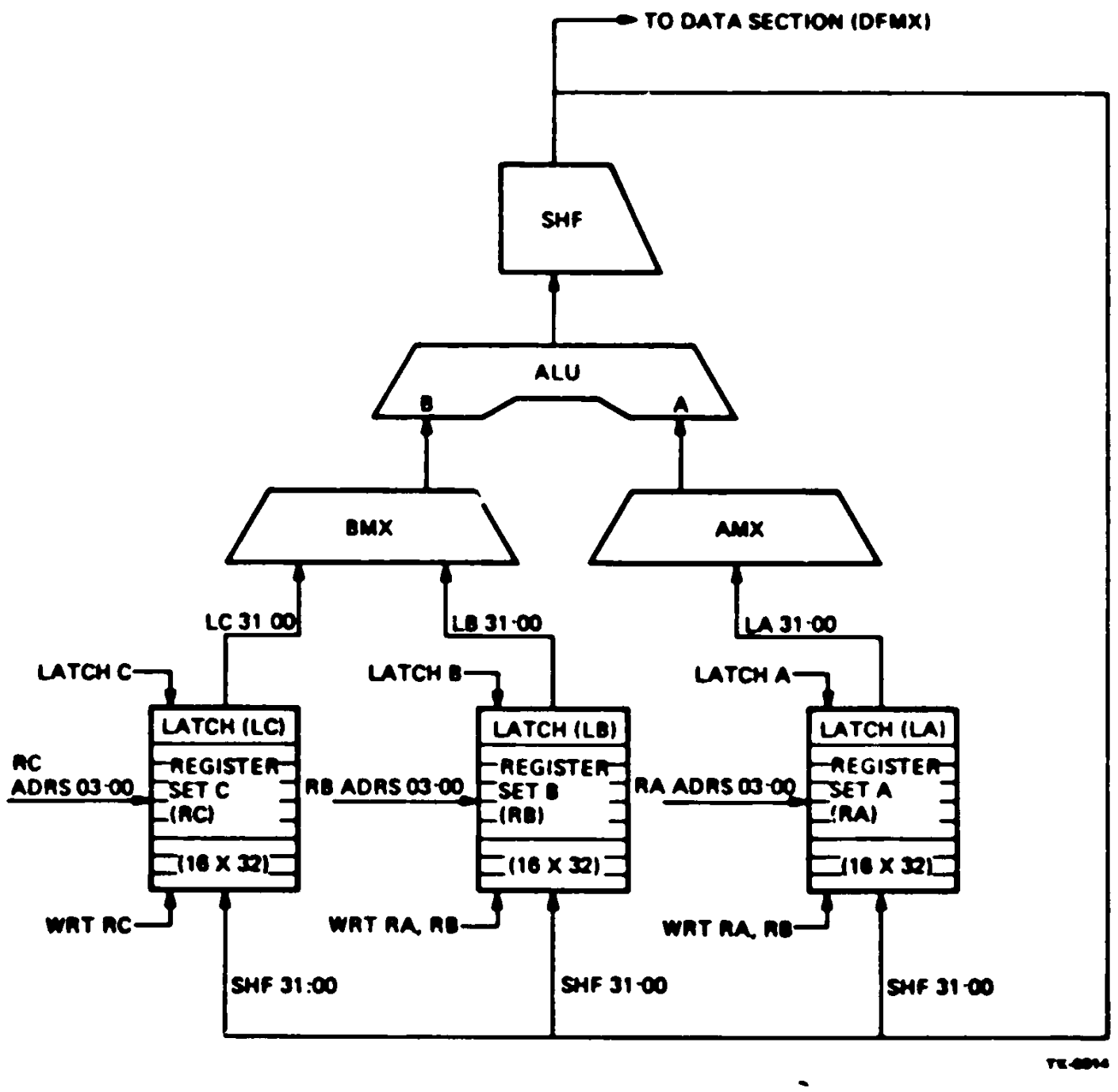


Figure 3-70 Scratchpad Register Logic

Register Set A (RA) and Register Set B (RB) – RA and RB provide a two-port temporary storage area for the 16 CPU general registers. These registers are used during addressing mode evaluations or for fast access storage during instruction execution. On a register read, the output latches (LA and LB) hold the contents of the selected RA or RB register for use by the arithmetic section. Using both ports allows fast access to the source register contents (LB to the BMX) and to the destination register contents (LA to the AMX) on the execution of register to register instructions.

All three register sets and their output latches are controlled by the USPO field of the CPU microword as shown in Table 3-50. Register Set C (RC) is addressed using a temporary register number (T0-TF) or by SC register bits (03:00). Register Sets A and B (RA and RB) are addressed by a general register number (R0-RF) or an address code number (ACN) that depends on the operating mode as shown in Table 3-51.

Table 3-50 USPO Field Selection of the Scratchpad Registers

USPO (06:04) and Range	USPO Bits (03)	(02)	(01)	(00)	Scratchpad Operation	Register Selection
00–05	0	X	X	X	No operation (NOP)	
06	0	1	1	0	Load LC	Address = SC (03:00)
07	0	1	1	1	Write RC, RB	Address = SC (03:00)
08–0F	1	A	C	N	Load LA, LB	Address selected by ACN value
10–17	1	0	R	N	Load LA	Address = general register (R0-R7)
18–1F	1	A	C	N	Write RA, RB	Address determined by ACN value
20–2F	0	R	N		Load LC	Address = temporary register (T0-TF)
30–3F	0	R	N		Write RC	Address = temporary Register (T0-TF)
40–4F	0	R	N		Load LA, LB	Address = general register (R0-RF)
50–5F	0	R	N		Write RA, RB	Address = general register (R0-RF)
60–6F	0	R	N		Load LA, LB	Address = general register (R1)
					Also Write RC	Address = temporary register (T0-TF)
70–7F	0	R	N		Load LC	Address = temporary register (T0-TF)
					Also write RA, RB	Address = general register (R1)

Table 3-51 Scratchpad Address Code Number (ACN)

ACN Hex Value	Native Mode RA Address	RB Address	Compatibility Mode RA Address	RB Address
00	SP1 R	SP1 R	SRC R	SRC R
01	SP1 R	SP2 R	DST R	DST R
02	SP2 R	SP1 R	DST R	SCR R
03	PRN	PRN	SRC R	SRC R
04	PRN plus 1	PRN plus 1	SRC R OR 1	SRC R OR 1
05	SC (03:00)	SC (03:00)	SC (03:00)	SC (03:00)
06	SP1 R plus 1	SP1 R plus 1	SRC R plus 1	SRC R plus 1
07	0	0	0	0

Native Mode Three register values are determined by the specifier 1 register (SP1 R), the specifier 2 register (SP2 R), and the previous register number (PRN) SP1 R is the register number of the operand specifier currently being evaluated by the instruction buffer control logic SP2 R is the register number of the operand specifier that follows the specifier currently being evaluated by the instruction buffer control logic

Compatibility Mode Two register values are determined by the source register (SRC R) and destination register (DST R) The SRC R and DST R numbers are defined by the register field of the current instruction

In either native or compatibility mode, the ACN value may also select SC (03:00) as the address source for the RA and RB sets, allowing the general registers to be addressed sequentially

USPO values 60-7F provide individual control of the RC and the RA/RB register sets in the same microinstruction, allowing the RC register to be written while the RA/RB registers are read or vice versa

RC register write operations are always longword data types. RA and RB register writes are context-dependent and are controlled by the UDT field of the CPU microword as shown in Table 3-52. When the UDT field is equal to 3, the instruction decode logic determines the data type to be used.

Table 3-52 UD Field Control of RA/RB Register Writes

UDT ⟨01:00⟩	Context
0	Longword (longword, quadword, floating, or double-floating)
1	Word
2	Byte
3	Instruction dependent (any of the above)

3.4.3.8 Register Log Stack (RLOG) and PC Save (PCSV) Register – The RLOG stack and PCSV register store the information needed to restart an instruction.

PCSV Register – The PCSV register is loaded with the lower eight bits of the program counter (PC ⟨07:00⟩) when an instruction is fetched. With this information, the 32-bit starting PC can be reconstructed if a a fault occurs. The high-order PC bits are saved in the instruction physical address (IPA) register of the translation buffer. Only the lower eight PC bits are saved in the PCSV register because no instruction is longer than 256 bytes.

RLOG Stack – The RLOG stack contains 16 × 32 bit locations and keeps a record of changes made to the RA register set during the instruction sequence (register autoincrement or autodecrement, for example). If an instruction generates a memory management fault, the information stored in the RLOG stack allows the general registers to be reloeded to their original states after the condition is serviced so that the instruction can be restored or restarted.

Each RLOG entry, as shown in Figure 3-71, contains the following information in the format: the address of the modified RA register, the constant used in the operation, and the ADD/SUBTRACT bit.

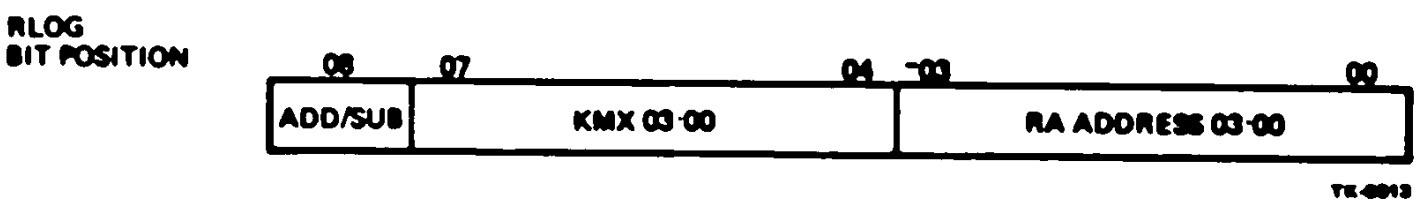


Figure 3-71 RLOG Entry Format

On each instruction fetch, a pointer into the stack and an RLOG empty flag are asserted. When the microcode fault routine reads the RLOG stack, the pointer is decremented and the next entry in the stack is available. The RLOG stack is written when the UALU field of the CPU microword specifies an RLOG update. If the ULAU field equals 6, the operation selected is A PLUS B (RLOG UPDATE) and RLOG bit (08) is set to a 1 (ADD). If the UALU field equals 1, the operation selected is A MINUS B (RLOG UPDATE) and RLOG bit (08) is set to a 0 (subtract). The RLOG stack is then read when the UMSC field of the CPU microword equals 7 (READ RLOG). The current value is read from the stack and the pointer decremented at the end of the microinstruction.

3.4.4 Exponent Section

The exponent section shown in Figure 3-72 processes floating-point exponent values in parallel with the fraction processing taking place in the arithmetic and data sections. It also generates values for the shift count (SC) register used by the arithmetic and data sections.

The exponent ALU (EALU) is the processing unit for the exponent section. The ten-bit data path consists of an eight-bit exponent with a two-bit overflow/underflow code.

A packed floating-point number is formed by the arithmetic section ALU B input multiplexer (BMX) in the format shown in Figure 3-73. The exponent comes from the EALU and the fraction values come from the D register in the data section. The destination fraction sign (SD) is determined by the USGN field of the microinstruction.

The exponent is written back to the shift count multiplexer (SMX) in the exponent section through ALU bits (14:07). The entire floating point number is then transferred through the arithmetic section ALU and ALU shifter (SHF) to the data section format multiplexer (DFMX) which unpacks the floating-point number and reassembles the fraction for processing.

3.4.4.1 Exponent Arithmetic Logic Unit (EALU) - EALU functions are selected by the UEALU field of the microinstruction as shown in Table 3-53.

The EALU output is asserted to the negative absolute value ROM (NABS ROM) and exponent multiplexer (EXP MUX). When the UALU field equals 7, the EXP MUX selects the NABS ROM, which provides the correct shift value for floating-point arithmetic alignment as demonstrated by the following example of an EALU function.

ADD 7 plus 12 (decimal)

When normalized, the two operands represent the following binary floating-point numbers:

$7 = 111 \times 2^3$	Exponent = 3, fraction = 111
$12 = 11 \times 2^4$	Exponent = 4, fraction = 11

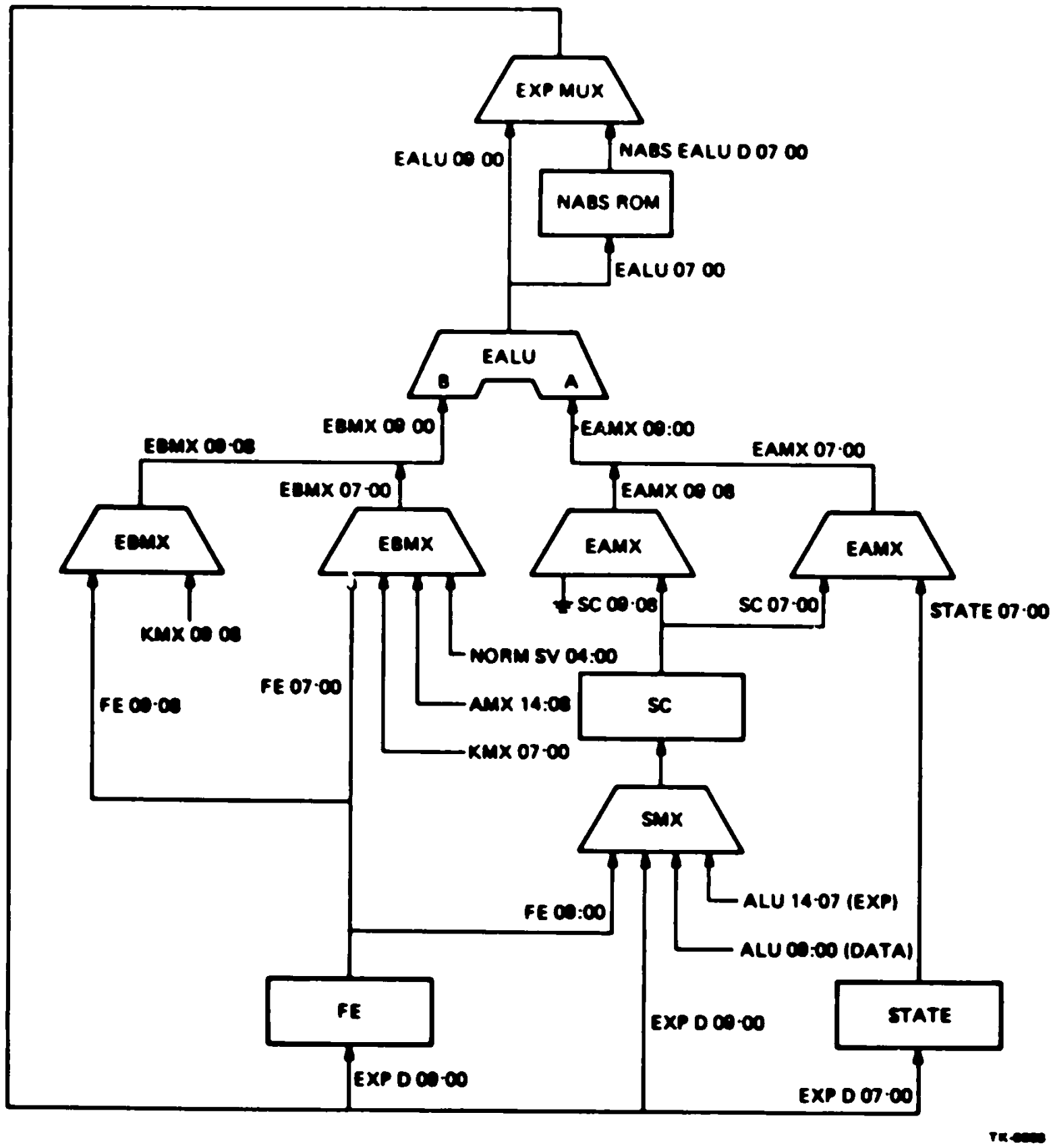


Figure 3-72 Exponent Section Logic

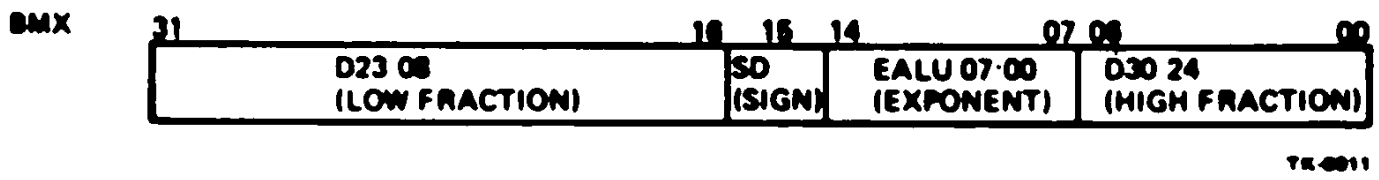


Figure 3-73 BMX Data in Packed Floating-Point Format

Table 3-53 UEALU Field Control of the EALU

UALU (02:00)	EALU Select (S3:S0)	MODE 1 = Logic 0 = Arithmetic	C Input 1 = With Carry	EALU Function
0	F	1	1	A
1	E	1	1	A OR B
2	B	1	1	A AND B
3	A	1	1	B
4	9	0	0	A plus B
5	6	0	1	A minus B
6	0	0	1	A plus 1
7	6	0	1	A minus B (EXP MUX selects NABSV ROM)

To perform the floating-point addition, the exponents of the operands must be equal or aligned, which requires a shift of the fraction with the smaller exponent. To begin execution, the operands are stored as follows:

- The source exponent (3) is stored in the FE register.
- The source fraction (111) is stored in the D register
- The destination exponent (4) is stored in the SC register.
- The destination fraction (11) is stored in the Q register.
- Both fractions are also stored in temporary registers.

The EALU function, SC minus FE, results in a positive 1, indicating that the fraction with the smaller exponent must be shifted one bit position to the right (a negative shift) to align the fractions. (To align the fractions for a negative result, a left shift would be required.)

In the above example, the positive 1 results in the NABS ROM being selected by the microcode, and the output is a negative 1 in two's complement form (FF). The output of the EXP MUX is then 3FF because the two most significant bits are hardwired as 1s.

Only the D register contents can be shifted and, in this example, the smaller fraction is stored in the Q register. The contents of the Q register are first loaded into the D register, which is then shifted by the value resulting from SC minus FE (1 bit). The fraction with the larger exponent is then loaded into the Q register from the temporary storage register and the fractions are added.

3.4.4.2 EALU A Input Multiplexer (EAMX) - The EAMX supplies the A inputs to the EALU and selects either the state register or shift count (SC) register. The state register is selected when the UMSC field of the CPU microword is equal to 5. Otherwise, the shift count register is selected.

3.4.4.3 EALU B Input Multiplexer (EBMX) - The EBMX supplies the B inputs to the EALU and selects either the floating exponent (FE) register, the AMX from the arithmetic section, the normalized shift value (NORM SV) from the data section, or the constant multiplexer (KMX) from the arithmetic section.

When exponents are being processed, the EBMX receives the exponent from the FE register or from the exponent field of the AMX (bits (14:07)). The constant inputs (KMX (07:00)from the arithmetic section) or the shift value inputs (NORM SV (04:00) from the data section) are selected to update the shift count (SC) register. The constant input is also used to set or clear flags in the state register.

The shift value generated in the data section (NORM SV) is the number of left shifts it takes to normalize the contents of the D register. The fraction part of a floating-point number, for example, is normalized by left shifting the D register contents until the most significant 1 of the data is in bit position (31).

The input selection to the EBMX is controlled by the UEBMX field of the CPU microword as shown in Table 3-54.

Table 3-54 UEBMX Field Control of the EBMX

UEBMX (01:00)	EALU B Input Multiplexer (EBMX) Data Selected
0	FE (09:00)
1	KMX (09:00)
2	AMX (14:07) (exponent field)
3	NORM SV (04:00)

3.4.4.4 Floating Exponent (FE) Register - The FE register holds exponent or temporary values for processing by the exponent section It is loaded with the output of the exponent multiplexer (EXP D ⟨09:00⟩) when the one-bit UFEK field of the CPU microword is equal to 1.

3.4.4.5 State Register - The state register contains eight flag bits generated by the microprogram that control main program flow A 16-way branch condition is created in the microsequencer by each four-bit group of state register bits.

A microinstruction sets or clears individual bits in the state register using EALU logic operations and constants from the data section KMX. The state register is loaded with the exponent multiplexer outputs (EXP D ⟨07:00⟩) when the UMSC field of the CPU microword is equal to 5.

3.4.4.6 Shift Count Multiplexer (SMX) - The SMX transfers data from the arithmetic section to the exponent section It provides the path for a ten-bit data field (ALU ⟨09:00⟩) or an eight-bit exponent (ALU ⟨14:07⟩) from the arithmetic section ALU to the shift count register of the exponent section.

It also selects the exponent multiplexer (EXP D ⟨09:00⟩) or floating exponent register (FE ⟨09:00⟩) as inputs for the shift count (SC) register. The exponent multiplexer causes the SC register contents to be incremented or decremented through the EALU. The FE register inputs also allow the contents of FE and SC to be swapped in a single CPU microinstruction.

The data type selected by the SMX is controlled by the USMX field of the microinstruction as shown in Table 3-55.

Table 3-55 USMX Field Control of the SMX

USMX ⟨01:00⟩	Selected Shift Count Multiplexer (SMX) Data
0	EXP ⟨D9:D0⟩
1	FE ⟨09:00⟩
2	ALU ⟨09:00⟩ (data field)
3	ALU ⟨14:07⟩ (exponent field)

3.4.4.7 Shift Count (SC) Register - The SC register is loaded from the shift count multiplexer (SMX ⟨09:00⟩) when the one-bit USCK field of the CPU microword is equal to 1. It is used for different functions in the areas of the CPU shown in Table 3-56.

Table 3-56 Shift Count (SC) Register Functions

Area of CPU	Shift Count (SC) Register Function
ID Bus Control	SC ⟨05:00⟩ select an internal processor register.
Data Section	SC ⟨09⟩ and SC ⟨04:00⟩ control the shift value in the data aligner (DAL).
Arithmetic Section	SC ⟨04:00⟩ control the bit mask generator. SC ⟨03:00⟩ address the scratchpad register sets.
Exponent Section	The SC register stores an exponent or data.

3.5 EXCEPTIONS AND INTERRUPTS

Exceptions and interrupts are CPU responses to service requests by system conditions or events. Both types cause the processor to leave normal program flow and enter a subroutine that services all conditions in priority order. Figure 3-74 shows the VAX-11/785 exception and interrupt structure which has the following forms.

Exceptions are generated in and are synchronous to the processor. They are initiated by the following types of internal processor conditions:

- *Traps* are detected by the microcode or CPU logic and serviced between instructions. Processing continues with the next instruction.
- *Faults* are detected by the microcode and serviced during the instruction. The instruction is backed up and restarted by the microcode after the fault is serviced.
- *Aborts* are detected by the microcode and serviced during the instruction. If the service routine can correct the condition, the microcode attempts to continue the instruction from the point it was aborted. If not, the process may be aborted or the system halted (system crash)

Interrupts consist of the following types of conditions that are generated internal or external to the processsor. They are asynchronous to the processor and are synchronized by the microcode.

- *Hardware* interrupts are initiated by the hardware and are serviced between instructions.
  - Internal hardware interrupts are requests for service by internal CPU events or conditions (such as a power failure or the interval timer).
  - External hardware interrupts are lower in priority than the internal requests. These are requests for service by external SBI devices (such as the UNIBUS or MASSBUS adapters).
- *Software* interrupts are initiated by software request from an internal CPU register. These interrupts are lower in priority than the hardware interrupts and are serviced only after certain instructions.

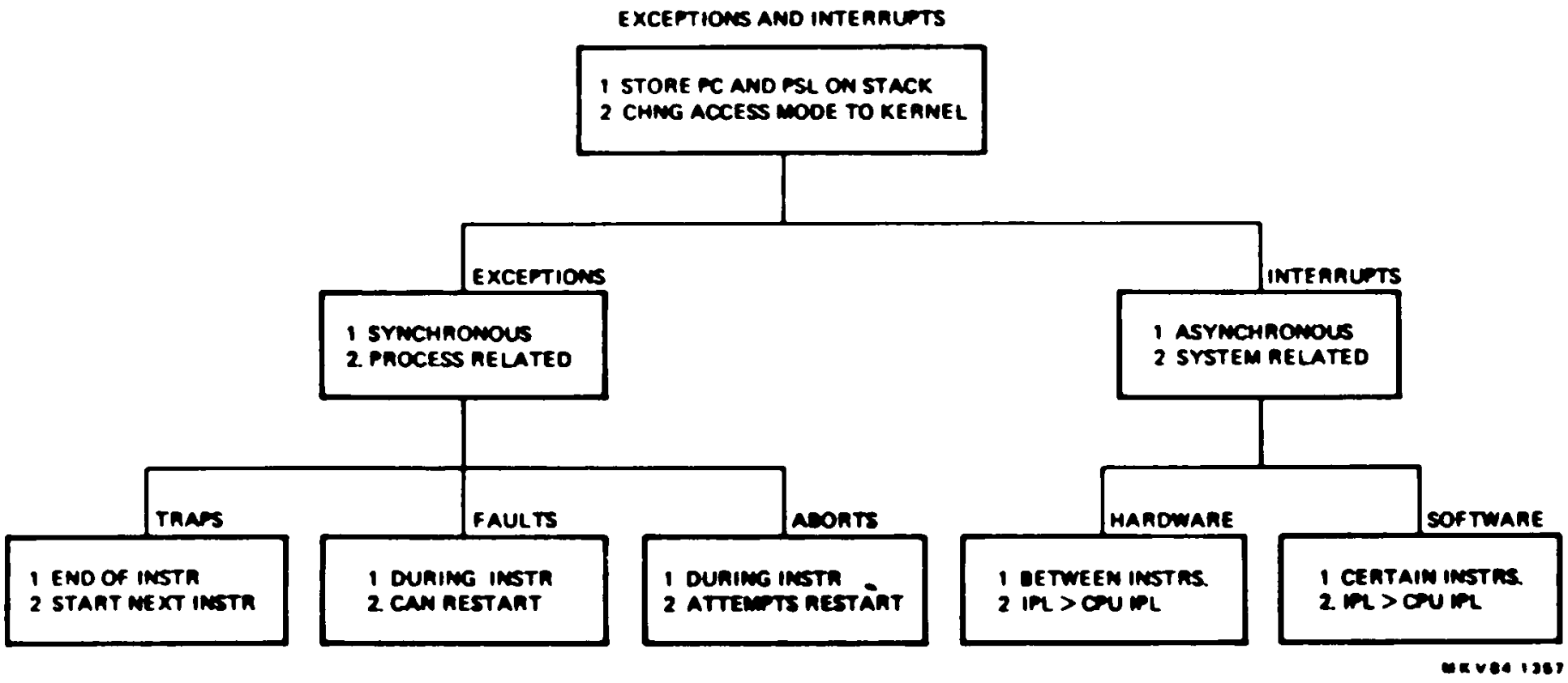


Figure 3-74 Exception and Interrupt Structure

3.5.1 General Description

The CPU microprogram (microcode) takes the following action when it initiates service for an exception or an interrupt

1. Saves the PC and PSL contents on the current stack (kernel or interrupt) and changes the processor access mode to kernel

2. Reads a vector address from the interrupting event or condition and determines the starting address of the service subroutine from the system control block (SCB)

3. Directs the CPU to enter the service subroutine for the condition

3.5.1.1 Exceptions Exceptions originate in and are synchronous to the CPU They handle conditions that occur during the execution of an instruction or series of instructions and are generally related to the current process Exceptions detect the following types of conditions

- *Traps* - The following conditions are detected by traps
  - Trace trap
  - Change mode trap
  - Arithmetic traps
  - Microtraps
- *Faults* - The following condition are detected by faults
  - Access violation
  - Translation not valid
  - Reserved addressing mode
  - Reserved operand (may be a fault or abort)
  - Reserved opcode
  - Compatibility mode (except for odd address abort)
  - Breakpoint
- *Aborts* - The following conditions are detected by aborts which require system-level evaluations by VMS

- Kernel stack not valid	- Aborts the process but not the system
- Interrupt stack not valid	- Aborts the process and halts the processor (crashes the system)
- Machine check	- Determines if the failure is recoverable and whether to abort the process
- Reserved operand	- May be a fault or abort
- Odd address	- Compatibility mode abort

3.5.1.2 Interrupts Interrupts are asynchronous to system operation and are synchronized by the microcode They handle systemwide events and conditions that belong to the system or to processes other than the one currently executing on the system

Interrupt Priority Level (IPL) - The CPU priority decode logic detects all hardware or software interrupt requests and grants service to the device or user with the highest asserted IPL However, the program is interrupted only when the processor IPL is changed to a lower level than the IPL of the highest request

Most exception service routines execute at the lowest IPL (IPL 0) However, there are a couple of exceptions representing serious system failures that change the CPU to the highest level (IPL 1F), preventing further interrupts until the condition has been serviced

Internal Hardware Interrupts - Hardware events such as a power failure or the interval timer are internal to the CPU They are higher in priority than the external hardware IPLs and each one has its own specific IPL number On an interrupt, the highest asserted IPL asserts its interrupt vector directly to the microcode logic

External Hardware Interrupts Devices external to the CPU assert a bus request signal through the arbitrator logic to the CPU priority decode logic

The microcode responds to the highest IPL and issues an interrupt summary read command to that level All SBI devices requesting an interrupt at that level respond with an interrupt summary response

The microcode determines the highest priority device and, when the CPU is able to field the interrupt, the arbitrator transmits a grant signal This device asserts its vector, which the microcode uses to assemble the starting address of the service subroutine

Software Interrupts Software IPLs are internal to the CPU and are lower in priority than the hardware IPLs They are generated by writing the software interrupt request register (SIRR) with the desired IPL which are then asserted to the CPU priority decode logic When the interrupt takes place, the highest asserted software IPL asserts a microvector address directly to the microcode logic

3.5.1.3 Compatibility Mode Exceptions and interrupts that occur in compatibility mode cause the processor to enter the native mode for service The following exceptions are reserved opcode faults that create a longword frame on the kernel stack consisting of the PC and PSL and a status longword that defines the type of fault

- PDP-11 reserved instruction
- BPT instruction
- IOT instruction
- EMT instruction
- TRAP instruction
- PDP-11 illegal instruction

3.5.1.4 Differences - Exceptions and interrupts have the following differences in the way they are generated and serviced

	Exceptions	Interrupts
1	Caused by the execution of the current instruction	Caused by system activity that may be independent of the current instruction
2	Synchronous, predictable conditions or events Tested or detected by the microcode	Asynchronous events external to processor control Detected by the interrupt logic and synchronized by the microcode
3	Usually serviced in the context of the process that produced the condition	Serviced independently of the currently executing process
4	The processor IPL is not usually changed when an exception is initiated	IPL is always changed when an interrupt is initiated
5	Service routines normally execute on a per-process stack (usually the kernel stack)	Service routines normally execute on a per-CPU stack (interrupt stack)
6	Enabled exceptions are serviced immediately, regardless of the current processor IPL	Interrupts are not serviced until the processor IPL drops below the IPL of the requesting interrupt
7	Only a few exceptions can be disabled If one of these events occurs while disabled, the exception is not initiated if it is subsequently enabled	If an interrupt condition occurs while the interrupt is disabled (the processor is at the same IPL or higher, or an interrupt enable bit is not set) an interrupt is initiated when the enabling conditions are met
8	The PSL previous mode field stores the mode of the CPU when the exception occurs	The PSL previous mode field is always kernel

3.5.2 Interrupts

The processor services an interrupt request at the end of an instruction or at defined points during the execution of a long, iterative instruction, such as the string instructions.

Table 3-57 lists the events and conditions that cause interrupts and lists their IPL and vector assignments. The three types of interrupts are as follows:

- Internal hardware interrupts
- External hardware interrupts
- Software interrupts

3.5.2.1 Interrupt Priority Levels (IPLs) – The processor has 31 IPLs that are divided into 16 hardware levels ((1F:10)) and 15 software levels ((0F:01)). User programs and most exception service routines execute at process level IPL 0 and interrupt requests are sampled by the CPU microcode during the execution of each machine instruction.

Software and hardware requests are clocked to the priority logic from the hardware interrupt register (HIR) and the software interrupt register (SIR) as shown in Figure 3-75. The priority encoder logic selects the highest requested interrupt level and generates an interrupt level active code (IPL ACT (04:00)) that is compared with the current processor IPL (bits (20:16) of the PSL). When the IPL of the highest interrupt request is greater than the current processor IPL (and no exceptions occurred during the instruction), INTR REQ is generated, causing a branch in the microcode flow that generates an interrupt vector from the IPL ACT code and initiates the interrupt service routine.

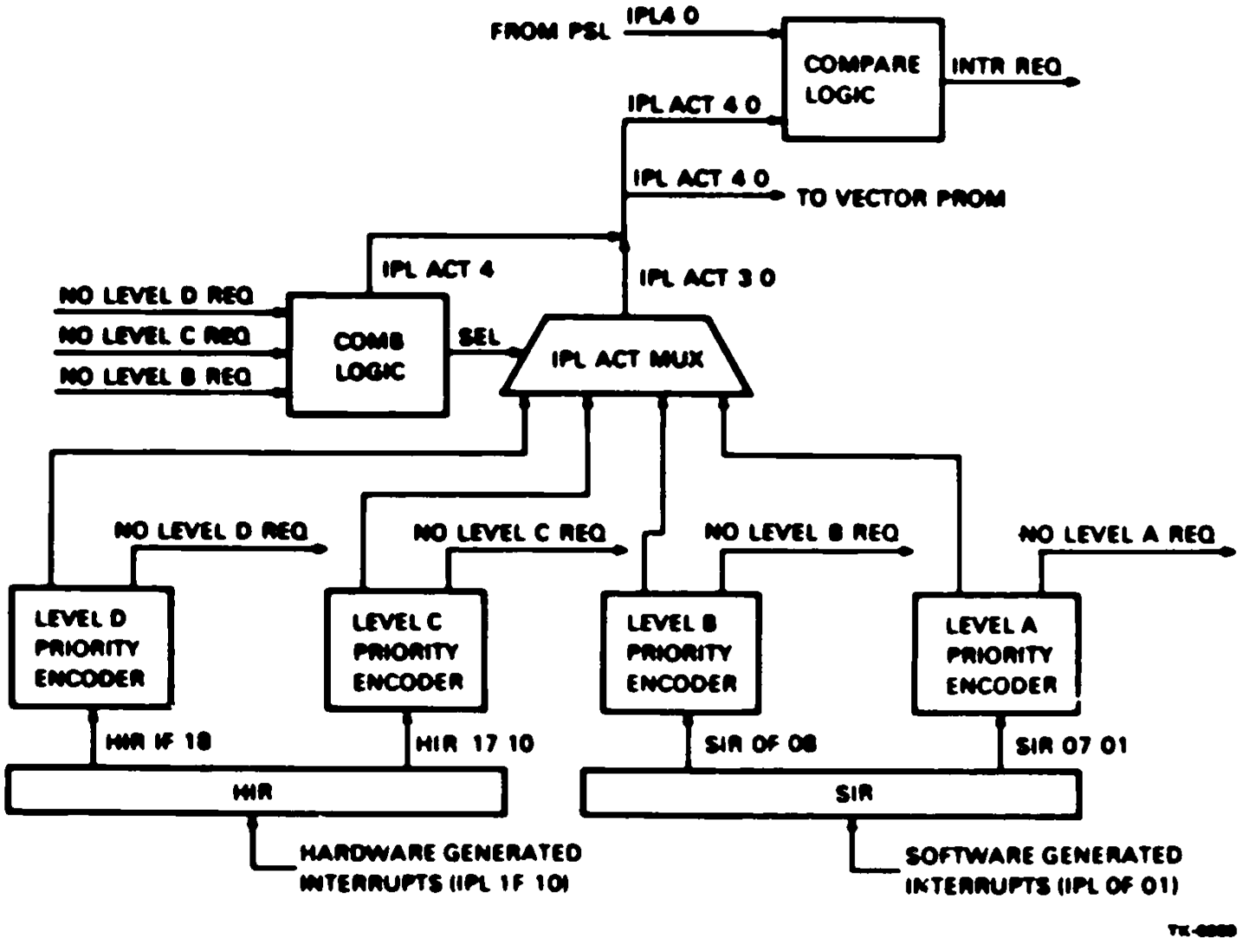


Figure 3-75 Interrupt Request Arbitration Logic

Table 3-57 Interrupt Conditions, IPL and Vector Assignments

IPL ACT	Interrupt Condition	Vector	Priority
1F	None assigned	-	Highest
1E	CPU Power Fail	0C	
1D	CPU Timeout	60	
1C	SBI Fault	5C	
1B	SBI Alert	58	
1A	CRD/RDS	54	
19	SBI SILO Compare	50	
18	Interval Timer	C0	
17	SBI REQ 7	1C0 TO 1FC	UNIBUS level BR7
16	SBI REQ 6	180 TO 1BC	UNIBUS level BR6
15	SBI REQ 5	140 TO 17C	UNIBUS level BR5
14	SBI REQ 4	100 TO 13C	UNIBUS level BR4
	CNSL Receive	F8	
	CNSL Transmit	FC	
13	None assigned	-	
12	None assigned	-	
11	None assigned	-	
10	None assigned	-	
0F	SIR0F	BC	
0E	SIR0E	B8	
0D	SIR0D	B4	
0C	SIR0C	B0	
0B	SIR0B	AC	
0A	SIR0A	A8	
09	SIR09	A4	
08	SIR08	A0	
07	SIR07	9C	
06	SIR06	98	
05	SIR05	94	
04	SIR04	90	
03	SIR03	8C	
02	SIR02 or AST DEL	88	
01	SIR01	84	Lowest
00	No interrupt	-	

3.5.2.2 Vectors A vector is a longword address in memory that points to the virtual starting address of an interrupt (or exception) service routine

Vector Address - Vector addresses are stored in a memory page called the system control block (SCB) The system control block base register (SCBB) holds the physical base address of the SCB page How an event is serviced is determined by the two lowest bits of the vector address as shown in Table 3-58 When bits (01:00) of a vector address contain a 0 or 1, bits (31:02) contain the virtual starting address of the service routine

Table 3-58 Event Control of Vector Address (01:00)

Vector Address (01)	(00)	Operation
0	0	Event is serviced on the kernel stack unless service is already in progress on the interrupt stack
0	1	Event is serviced on the interrupt stack The processor IPL is raised to 1F for an exception
1	0	Event is serviced by the CPU microcode and vector address bits (15 02) are used as a parameter.
1	1	The operation is a processor halt (HALT)

Interrupt Vector - An interrupt vector is the physical longword address of a location in the SCB It is formed by adding the received hardware generated vector bits (08 02) to the SCBB register as shown in Figure 3-76 (Bits (01:00) are zero because it is a longword address) The contents of the selected SCB location are the vector address (the starting address of the service routine)

3.5.2.3 Internal Hardware Interrupts - Bits (08:02) of a hardware vector are generated as a function of the interrupt priority active (IPL ACT) level and the status of the console receive and transmit lines (CREC INTR and CXMT INTR) as shown in Figure 3-77 IPL ACT (04 00) and the console lines select both combinational logic and a vector PROM that generate the signals VECTOR (08 02)

The combinational logic drives the external interrupt request lines for IPLs (17 14) However, both console interrupts also request on IPL 14 When the IPL ACT is 14 and the console interrupt lines are asserted without an external SBI REQ 4, EXT INTR REQ is disabled for internal interrupts and the VECTOR (08:02) lines are asserted by the vector PROM.

3.5.2.4 External Hardware Interrupts - Multiple interrupts can occur on each of the external IPLs (17 14) These levels correspond to SBI REQ (07:04) for devices connected to the SBI (through the UNIBUS or MASSBUS adapter, for example).

A separate vector is assigned to each device that can cause interrupts at each of the SBI levels and several devices at the same SBI level can request service at once However, the IPLs do not identify requesting devices The microcode issues an interrupt summary read command on the SBI in the format shown in Figure 3-78 and specifies the request level at which it is polling Devices asserting the request line specified by the interrupt level mask (B7 B4)), receive the interrupt summary read command and return an interrupt summary response, asserting a bit pair in the information field (B31 B00)

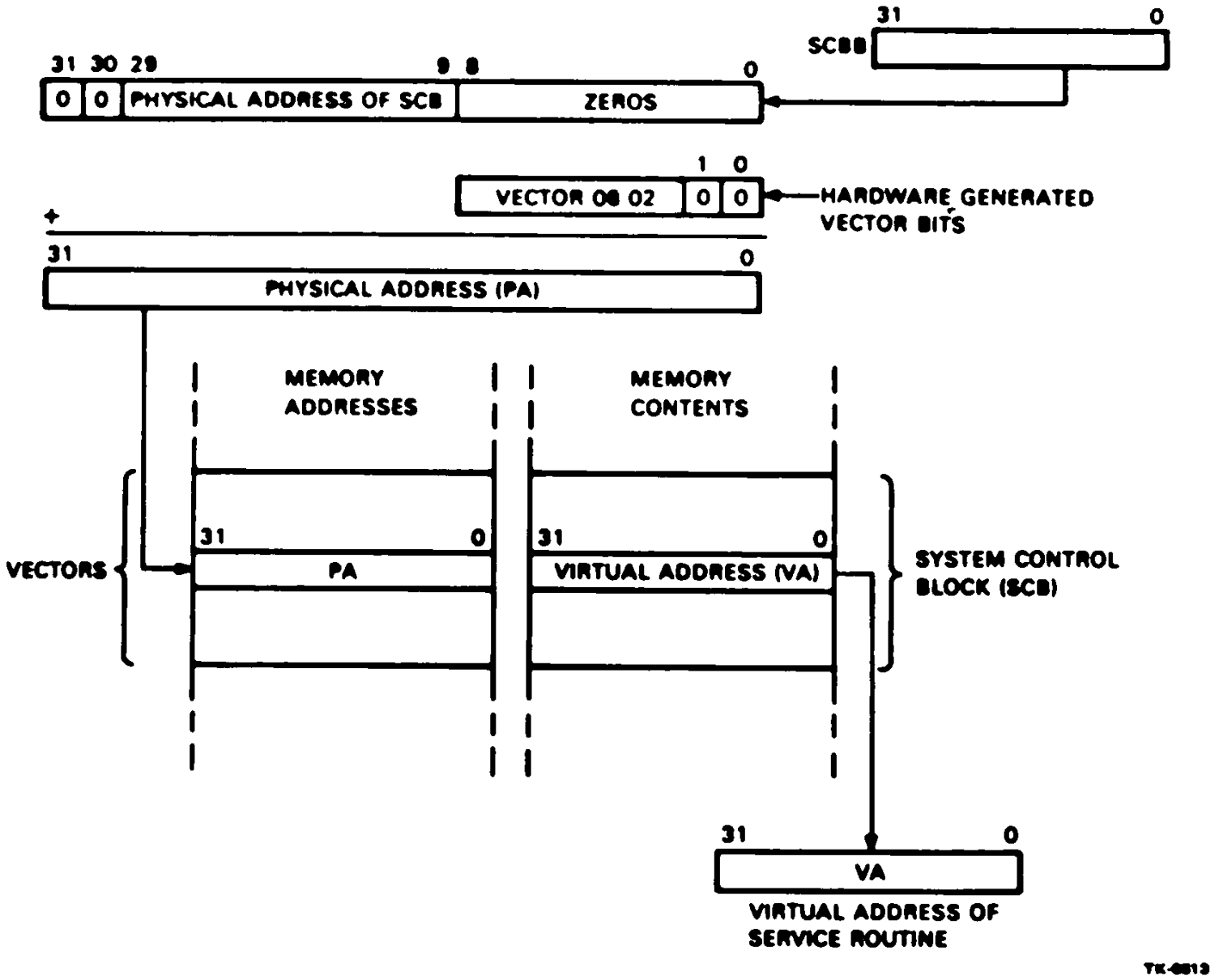


Figure 3-76 Forming the Interrupt Vector

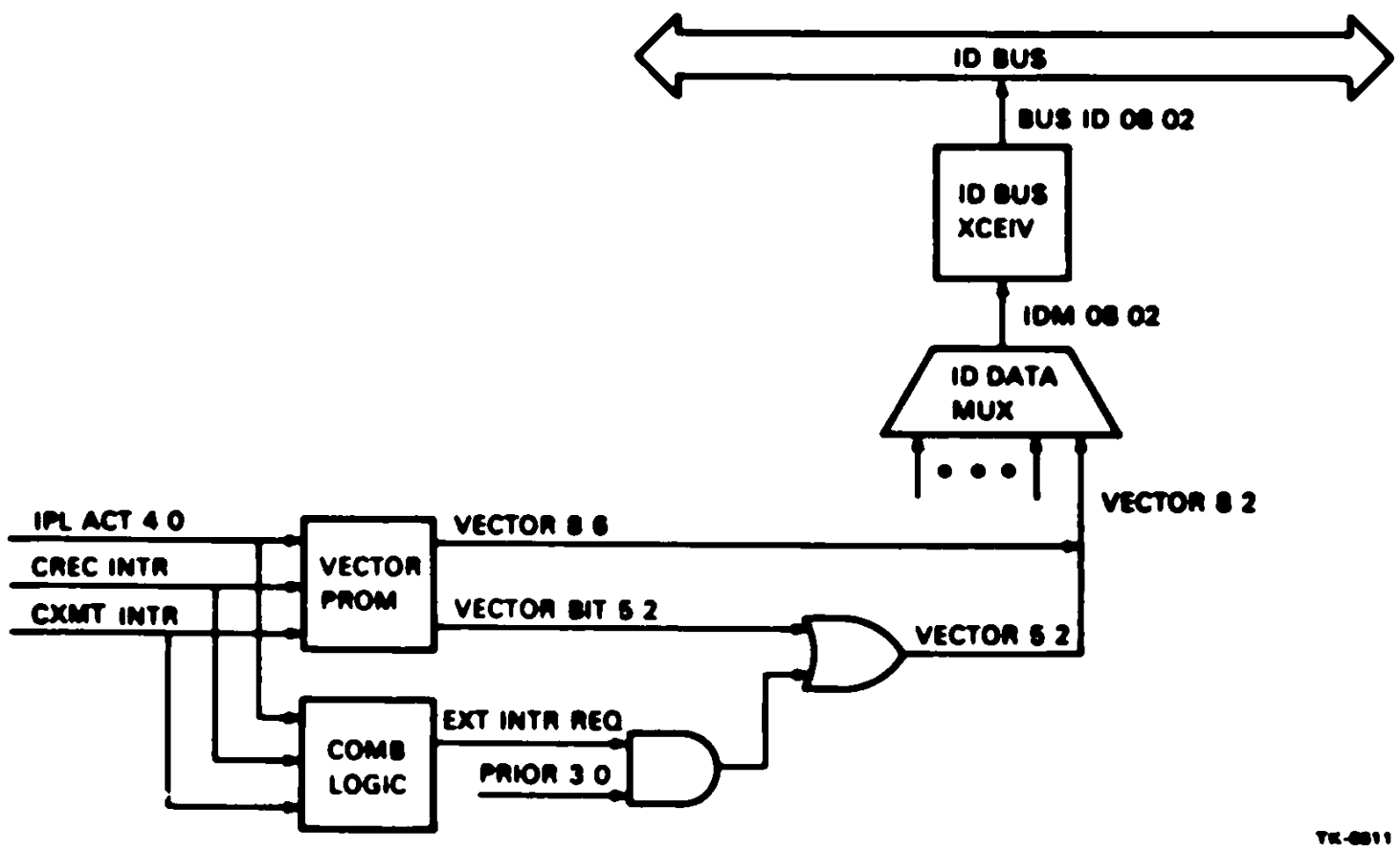


Figure 3-77 Generating Interrupt Vector Bits (08:02)

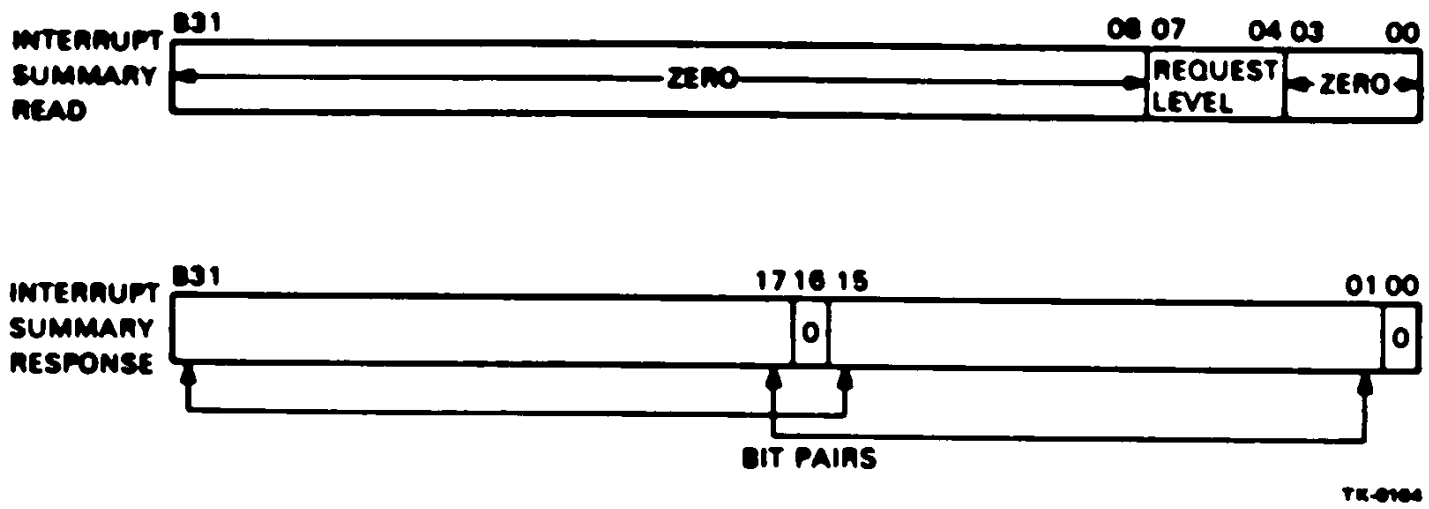


Figure 3-78 Interrupt Summary Read and Response Formats

The bit pairs are asserted in corresponding positions of the upper and lower half of the information field in order to maintain correct parity. The two bits asserted by the requesting device are equal to the device transfer request (TR) number and the TR number plus 16. A TR number is assigned to each SBI device or subsystem to establish the fixed priority access. Asserting a TR line in the interrupt summary response identifies the device that is interrupting at the polled interrupt request level.

The upper half of the information field ⟨B31:16⟩ is transferred to the interrupt logic on the internal data (ID) bus as shown in Figure 3-79. ID bus data ID ⟨31:16⟩ is input to priority/sum PROMs that encode the information for use in the vector register. The A or B priority bits specify the device with the lowest TR number (the highest priority) that interrupted at the polled request level. The output from PROM A is selected by the prior multiplexer if ID ⟨23:16⟩ is all zeros. The PRIOR VALID bit in the vector register, when set, indicates that ID ⟨31:16⟩ is not all zeros and that at least one device interrupted at the polled request level. The sum bits of PROMs A and B are added to form the number of 1 bits in the ID ⟨31:16⟩ field and are not used in the interrupt process.

The vector register is a read-only register on the internal data bus and operates in the format shown in Figure 3-80. Bits ⟨24:21⟩ (PRIOR ⟨03:00⟩) are used by the hardware to generate VECTOR lines ⟨05:02⟩ as shown in Figure 3-77. If the interrupt is decoded as an external request, PRIOR ⟨03:00⟩ are ORed with VECTOR PROM output bits ⟨05:02⟩, forming VECTOR ⟨05:02⟩. These lines are used with VECTOR ⟨08:06⟩, to point to the service routine unique to the interrupting device. The device service routine initiates a read transfer on the SBI to identify the type of interrupt requested by the device (a UNIBUS or MASSBUS device, for example). This information points to a subroutine that services that device. If multiple devices request interrupts at the same level, multiple interrupt summary read commands are issued by the microcode until all devices have been serviced.

3.5.2.5 Hardware Interrupt Register (HIR) – Figure 3-81 shows the HIR bit usage. The HIR supplies interrupt requests to the arbitration logic on interrupt priority levels ⟨1E:14⟩ as shown in Figure 3-75.

The hardware conditions that cause the interrupts are shown in Table 3-59.

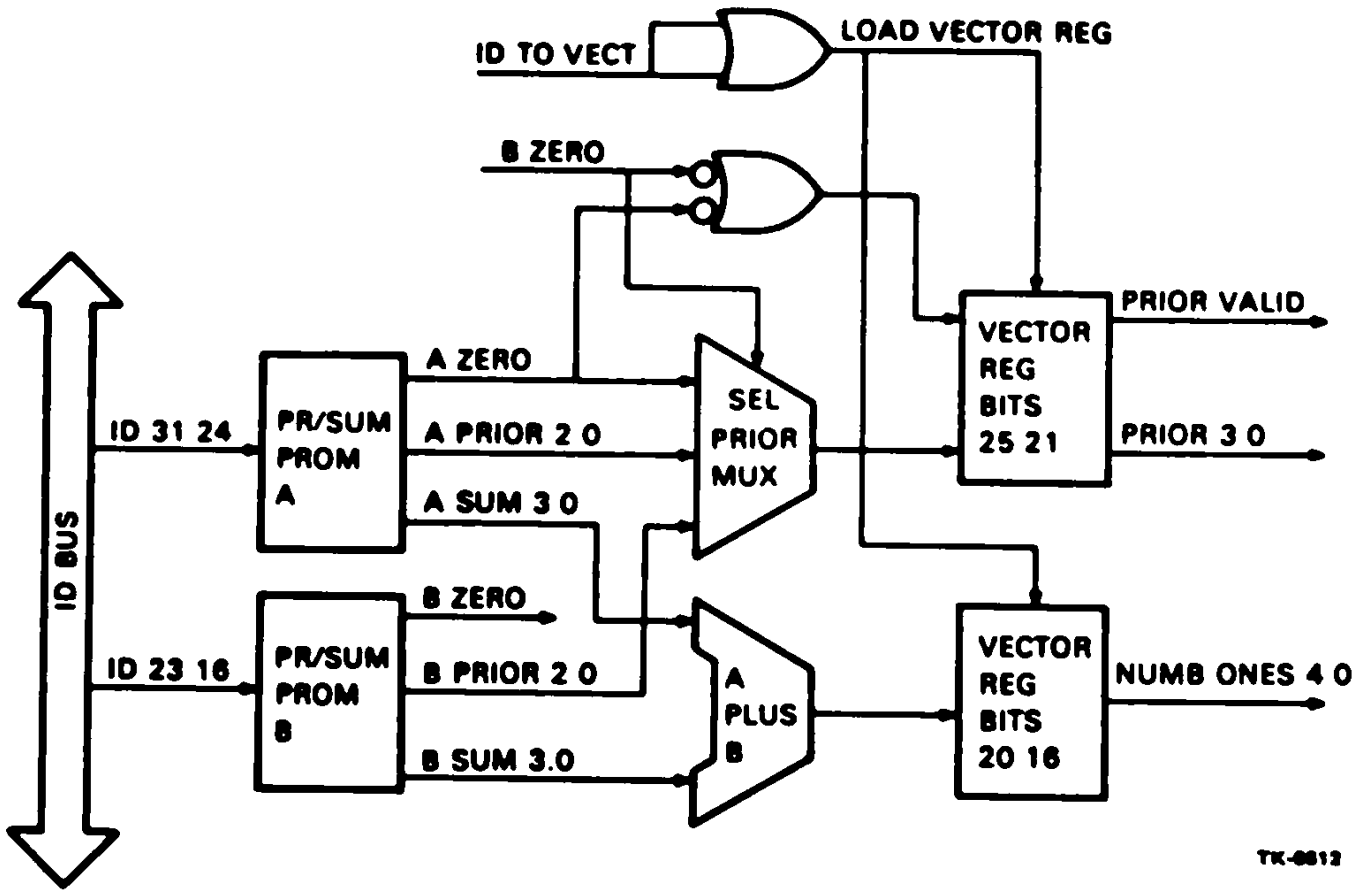


Figure 3-79 Generating Vector Register Bits ⟨25:16⟩

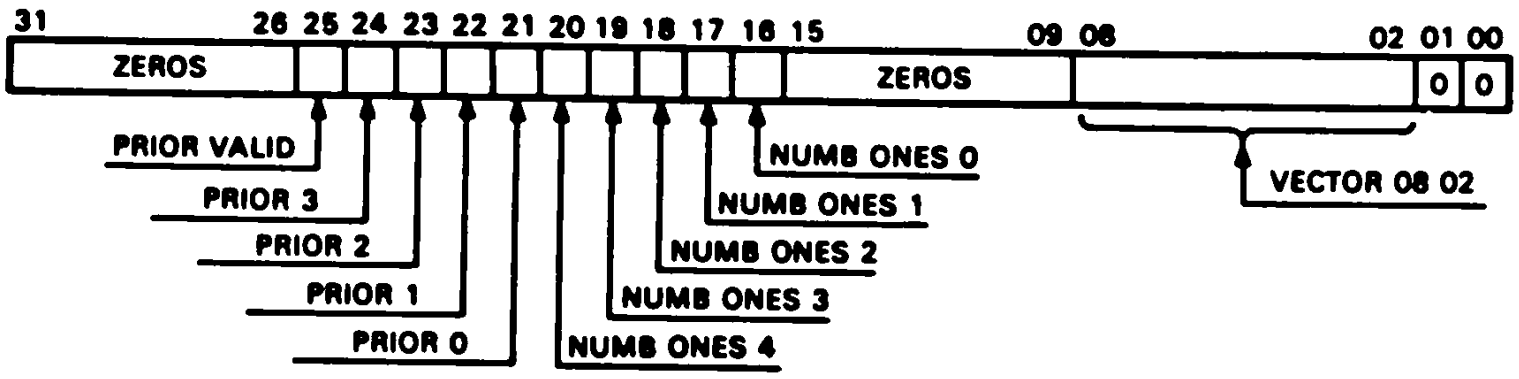


Figure 3-80 Vector Register

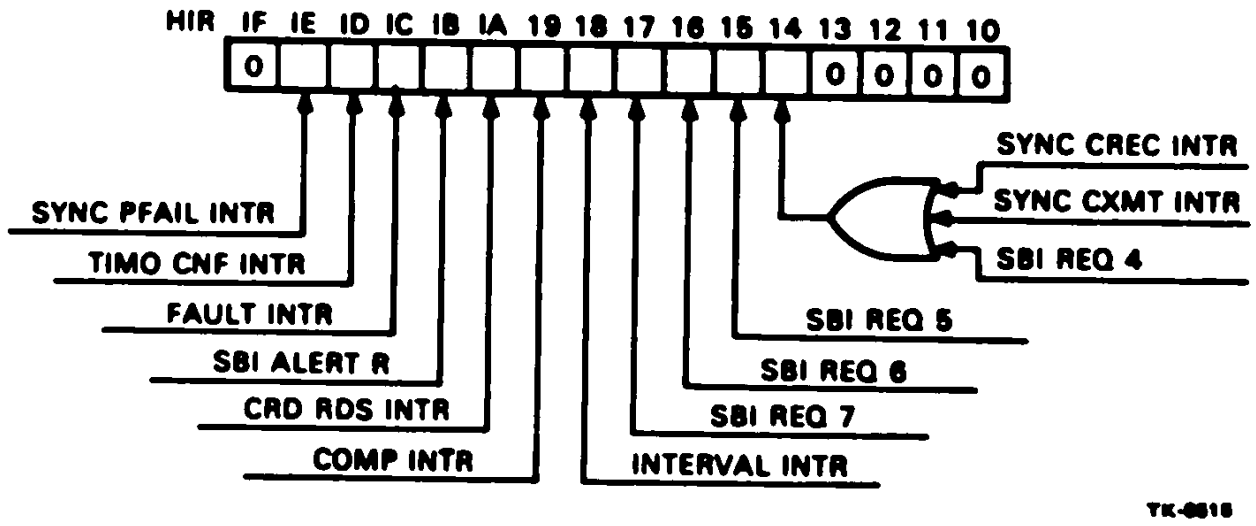


Figure 3-81 Hardware Interrupt Register (HIR)

Table 3-59 Hardware Interrupt Register (HIR) Bit Usage

IPL	Name	Description
1F	-	Not asserted by any device. The CPU is set to this level in the PSL for critical exceptions.
1E	SYNC PFAIL INTR	A power-fail interrupt occurs if the processor receives a power-fail warning (power supply AC LO) or a critical system element receives a power-fail warning (SBI FAIL or QBUS AC LO). The following critical system elements inside the system cabinet must be functioning before power-up routines are initiated. - LSI-11 microcomputer - Bootstrap ROM and main memory RAM logic - SBI termination - CPU and SBI clock circuits - All CPU logic
1D	TIMO CNF INTR	The CPU write timeout interrupt occurs if the processor initiates a write command and the destination does not respond with a confirmation within 512 SBI cycles. Write timeout interrupts do not necessarily occur during the instruction that initiated the write command. Write commands are stored in a buffer and the processor is allowed to continue while an SBI write cycle is pending.
1C	FAULT INTR	The SBI fault interrupt occurs if an SBI bus error is detected by any device on the bus including the processor. If the processor detects a fault condition that prevents the completion of a read cycle, an exception condition is also generated. (The fault interrupt enable bit must be set in the SBI fault/status register for this interrupt to take place.)
1B	SBI ALERT	Occurs on an interrupt by a device that does not contain SBI interrupt request sequencing logic. Events that causes this interrupt may be a device power failure or power-up, or when environmental conditions such as overtemperature are detected. The SBI alert line is asserted with the logical OR of the alert status bits in the device configuration register.

Table 3-59 Hardware Interrupt Register (HIR) Bit Usage (Cont)

IPL	Name	Description
1A	CRD/RDS INTR	The corrected read data (CRD) interrupt is asserted if the received read data has been corrected by main memory. The read data substitute (RDS) interrupt is asserted if uncorrected read data is received on a read cycle to the instruction buffer. (The CRD/RDS interrupt enable bit must be set in the SBI error register for these interrupts to take place.)
19	COMP INTR	The SBI silo compare interrupt occurs when certain fields of the SBI bus match signals are specified in the SBI comparator register. This interrupt occurs only if the silo lock interrupt enable bit is set in the SBI comparator register.
18	INTERVAL INTR	This interrupt occurs when the interval count register overflows and can only be initiated if enabled in the clock control status register.
17-14	SBI REQ ⟨07:04⟩	External interrupts occurring on SBI request levels ⟨07:04⟩ are serviced on IPLs ⟨17:14⟩. These interrupts occur on device completion, device errors, and on important device status changes.
14	SYNC CREC INTR	The console terminal receive interrupt occurs when the Done bit (RX DNE) is set in the console receive control/status register (RXCS). (The receive interrupt enable bit in the RXCS register must be set for the interrupt to take place.)
14	SYNC CXMT INTR	The console terminal transmit interrupt occurs when the Ready bit (TX RDY) is set in the console transmit control/status register (TXCS). (The transmit interrupt enable bit in the TXCS register must be set for the interrupt to take place.) The receive interrupt has higher priority than the transmit interrupt.

3.5.2.6 Software Interrupts – Interrupt priority levels ⟨0F:01⟩ are reserved for software use. Software can force an interrupt by executing the instruction MTPR SRC, #SIRR. This move to processor register instruction moves the contents of the source register to the software interrupt request register (SIRR). The SIRR is accessible in processor register space as register number 14 (hex). It is a write-only, four-bit register formatted as shown in Figure 3-82.

Figure 3-82 Software Interrupt Request Register (SIRR)

Executing MTPR SRC, #SIRR requests an interrupt at the level specified by the low four bits of the source register (SRC ⟨03:00⟩). When a software interrupt request is made, it is cleared by the hardware when the interrupt is taken. If SRC ⟨03:00⟩ is greater than the current IPL, the interrupt occurs before execution of the instruction. If SRC ⟨03:00⟩ is less than or equal to the current IPL, the interrupt does not occur until the IPL is lowered to less than SRC ⟨03:00⟩. If there are higher-level IPLs pending, they are serviced first.

Pending software interrupts are held in the software interrupt summary register (SISR). The SISR is read/write processor register 15 (hex) which is formatted as shown in Figure 3-83. The SISR contains 1s in the bit positions corresponding to the levels on which software interrupts are pending.

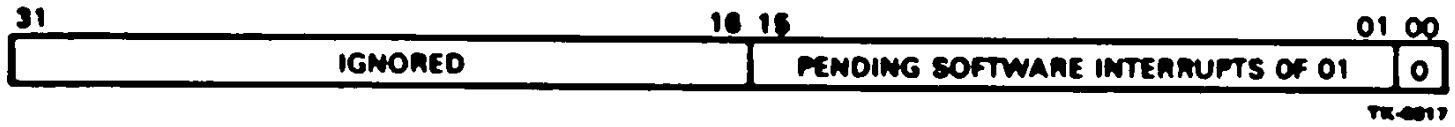


Figure 3-83 Software Interrupt Summary Register (SISR)

When a software request is made by a write to the SISR, the CPU microcode interprets it as a bit set operation to the SISR. The mask generator in the data paths decodes the request level in the SIRR to a single bit designation in the SISR. The SISR can be written directly by the instruction MTPR SRC, #SISR. This is not the normal method of making software interrupt requests. However, it is useful for clearing the software interrupt system and for reloading the system after a power failure.

Bit ⟨02⟩ of the SISR is also set if an asynchronous system trap (AST) is asserted at interrupt priority level 2 (IPL ⟨02⟩). During the execution of a return from exception or interrupt (REI) instruction, the microcode compares the value in the current mode field of the PSL image (bits ⟨25:24⟩) with the asynchronous system trap level (ASTLVL) in the ASTR register. If the current mode is a greater value (lesser priority) than the three-bit ASTLVL, an asynchronous system trap (AST) is asserted at interrupt priority level 2. The microcode sets bit ⟨02⟩ in the SISR before completing the execution of REI. The asynchronous system trap level register (ASTR) is a three-bit, read/write processor register 13 (hex), which is formatted as shown in Figure 3-84.

Figure 3-84 Asynchronous System Trap Level Register (ASTR)

3.5.2.7 Software Interrupt Register (SIR) – The software interrupt register shown in Figure 3-85 is an internal data (ID) bus register at ID bus address 0E (hex) and is located on the interrupt control (ICL) module.

Bits (15:01) of the SIR are read/write in processor register space and are in the same format as bits (15:01) of the SISR. When the SISR or SIRR is specified on an MTPR instruction, the SIR is clocked with data from ID bus (15:01). SIR (15:01) then specify which software interrupt requests are being made and bits (20:16) (IPL ACT (04:00)) indicate the highest pending priority interrupt level.

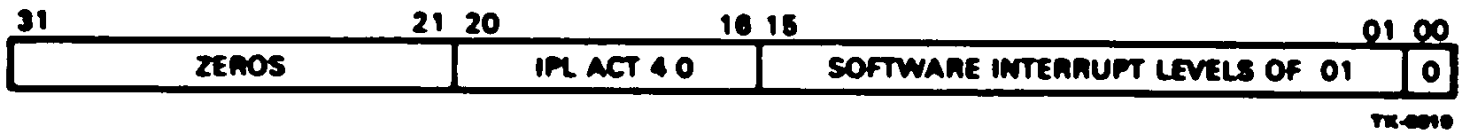


Figure 3-85 Software Interrupt Register (SIR)

3.5.3 Exceptions

Exceptions are the notification of events that are mainly relevant to currently executing process and normally invoke software in the context of the current process. Exceptions occur in the middle or at the end of the instruction during which the exception conditions were detected. The PSL and PC of the instruction, or the PC of the next instruction, are then pushed onto the kernel or interrupt stack. Also, up to 16 longwords of exception parameter information may be pushed onto the stack.

The processor's IPL is generally not changed by exceptions. However, two exception conditions, kernel stack not valid and machine check faults, raise the processor IPL to the highest priority level 1F (hex). Exceptions are grouped into one of the following three types, depending on when the exception condition occurs and how they leave the general register and memory states.

- *Trap* – Occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. Arithmetic traps are the only exceptions that can be disabled using the BISPSW and BICPSW instructions.

- *Fault* – Occurs in the middle of an instruction, leaving registers and memory in a state that allows correction of the fault conditons. Restarting the instruction gives the correct results. The PC saved on the stack is the address of the instruction in which the fault was detected.

- *Abort* – Occurs in the middle of an instruction and may leave registers and memory in a state from which the instruction can not be correctly completed or restarted. Also, the PC saved on the stack may not point to the beginning of the next instruction.

Exceptions modify microprogram flow in the microsequencer (USC) branch logic as described in Section 3 2 2 2, using one of the following two microtest methods

- *Microbranches* The microcode tests for microbranch conditions during the normal execution of all microinstructions These conditions are tested by specific branch enable values specified by the UBEN field of the microword If an exception condition exists, it directly modifies the next microaddress from the USC branch multiplexers

  Microbranches are also performed from service bits monitored by the instruction decode logic When the subroutine (USUB) field of the current microword is equal to 3, the instruction decode logic asserts the lower eight bits of the next microaddress. If any exception conditions are present, the service bits modify the next microaddress

- *Microtraps* The microcode tests for microtrap conditions during the normal execution of certain other microinstructions generating microtrap vector bits. A microtrap condition forces microcode flow to the service routine pointed to by the vector and pushes the microprogram counter (UPC) onto the microstack so that the program can continue after the error is serviced.

  When UBEN (04:00) are equal to 10 (hex), vector bits UTRAP VECT (03:00) are asserted as the lower four bits of the next microaddress from the USC branch multiplexers. The hardwired upper bits of the microaddress form the rest of the vector

3.5.3.1 Exception Vectors - Microservice routines are selected by the microbranch address or microtrap vector Each microservice routine generates the correct exception vector and exception codes using a set of constants specified by the UKMX field of the microword. The service routine also uses information stored in the processor registers to assemble the parameters to be saved on the stack. Each exception, class, vector assignment, and the method by which each exception condition is detected is shown in Table 3-60.

Table 3-60 Exception Conditions and Assigned Vectors

Exception Condition	Vector	Type	Detection Function
Machine check	04	Fault/Abort	Ubranch/Utrap
Kernel stack not valid	08	Abort	Ubranch
Reserved DIGITAL opcodes and privileged instructions	10	Fault	Ubranch
Reserved customer opcodes	14	Fault	Ubranch
Reserved operands	18	Fault/Abort	Ubranch/Utrap
Reserved addressing modes	1C	Fault	Ubranch
Access control violation	20	Fault	Utrap/Ubranch
Translation not valid	24	Fault	Ubranch
Trace pending (TP)	28	Fault	Ubranch
Breakpoint instruction (BPT)	2C	Fault	Ubranch
Compatibility mode program error	30	Trap/Abort	Ubranch
Arithmetic trap	34	Trap	Ubranch
CHMK opcode	40	Trap	Ubranch
CHME opcode	44	Trap	Ubranch
CHMS opcode	48	Trap	Ubranch
CHMU opcode	4C	Trap	Ubranch

3.5.3.2 **Serious System Failure Exceptions (Aborts)** - The following paragraphs briefly describe the exceptions that cause the IPL of the CPU to be raised to 1F (hex) and may cause the process to be aborted or the CPU to be halted

Kernel Stack Not Valid - This exception indicates that the kernel stack was not valid when the processor pushed information onto it during the initiation of an exception or interrupt This may be an indication of a stack overflow or other executive software error The attempted exception is changed to an abort that uses the interrupt stack, the IPL is raised to 1F (hex), and no additional parameters are pushed onto the interrupt stack

Interrupt Stack Not Valid - This exception indicates that the interrupt stack was not valid or that a memory error occurred when the processor pushed information onto the interrupt stack during the initiation of an exception or interrupt No further interrupt requests are acknowledged by the processor.

Machine Check - This exception indicates that an internal processor error was detected. The IPL is raised to 1F (hex) and, in addition to the PC and PSL, parameters are pushed onto the stack as longwords that depend on the type of machine check encountered. The last longword pushed specifies the number of additional bytes pushed, excluding the PC, PSL, and count longword The information pushed is logged for analysis by field service and allows the software to decide whether or not to abort the process.

On any machine check, the error handling microcode attempts to log the following information. It normally appears on the stack as shown in Table 3-61 However, if a double-error halt occurs, the operator can find the same information in the ID bus temporary registers.

Table 3-61 Machine Check Parameter Locations

Data	Memory Location	ID Bus Location
Byte count	(SP)	none
Summary parameter	(SP)+4	T0(30)
CPU error status	(SP)+8	T1(31)
Trapped UPC	(SP)+12	T2(32)
VA/VIBA	(SP)+16	T3(33)
D register	(SP)+20	T4(34)
TB ERR 0	(SP)+24	T5(35)
TB ERR 1	(SP)+28	T6(36)
Timeout address	(SP)+32	T7(37)
Parity	(SP)+36	T8(38)
SBI error	(SP)+40	T9(39)
PC	(SP)+44	none
PSL	(SP)+48	none

The summary parameter is a longword and byte 1 of the longword is a flag. It is non-zero if a CP timeout or CP error confirmation interrupt was pending at the time the machine check occurred. Byte 0 identifies the type of machine check as shown in Table 3-62.

Table 3-62 Machine Check Identification in Summary Parameter Byte 0

Machine Check Code	Exception Condition
Faults	
00	CP read timeout or error confirmation
02	CP translation buffer parity error
03	CP cache parity error
05	CP read data substitute
0A	IB translation buffer parity error
0C	IB read data substitute
0D	IB read timeout or error confirmation
0F	IB cache parity error
Aborts	
F0	CP read timeout or error confirmation
F1	Control store parity error
F2	CP translation buffer parity error
F3	CP cache parity error
F5	CP read data substitute
F6	Microcode "not supposed to get here"

The control store parity error is detected by a microtrap. The remaining exception conditions are detected by microbranches during instruction buffer cycles and microtraps during data path cycles.

Reserved Operand - This exception may result in a fault or abort as described below in Section 3.5.3.3.

Odd Address - This exception takes place if the process attempts to access an odd address in the compatibility mode The CPU is changed to the native mode to service the condition.

3.5.3.3 Exceptions During an Operand Reference (Faults) - The following exceptions are detected during operand evaluation.

Access Violation - This exception occurs when the process attempts a reference that is not allowed in the access mode at which the process is operating (protection violation). The exception also occurs if the referenced virtual address is beyond the end of the page table (length violation). If the entry is in the TB, the protection violation is detected by a microtrap during data path cycles and by a microbranch during instruction buffer cycles Otherwise, it is detected by a microbranch. The length violation is detected by the microbranch function

Translation Not Valid This exception occurs when a read or write reference is attempted through an invalid page table entry The exception is detected by a microbranch.

Reserved Addressing Mode This exception occurs on the attempted use of certain prohibited addressing modes and no additional parameters are pushed on the stack. Conditions causing this exception are shown in Table 3-63.

Table 3-63 Reserved Address and Mode Exceptions

Addressing Mode	Exception Condition
Short literal	Modify, destination, address source, or within index mode
Register	Address source or within index mode
Index	Within index mode, or with PC as index

Reserved Operand – This exception indicates that the operand accessed has a format reserved for future use by Digital Equipment Corporation. No additional parameters are pushed on the stack and the PC is backed up to point to the opcode The service routine determines the type of operand by examining the opcode using the stored PC Only changes made as a result of the instruction fetch or operand specifier evaluation can be restored Some instructions are therefore not restartable and the exception is an abort rather than a fault The PC is always properly restored unless the instruction attempts to modify it in a manner that yields unpredictable results The PSL, other than the FPD and TP bits, is not changed except for the condition codes, which are unpredictable

The following types of events cause a reserved operand exception and determine whether it results in a fault or in an abort.

- A floating-point number that has the sign bit set and the exponent as a zero except in the POLY table (fault)
- A floating-point number that has the sign bit set and the exponent as a zero in the POLY table (abort)
- POLY degree too large (fault)
- Decimal string too long (fault)
- Invalid digit in CVTTP or CVTSP (fault)
- Bit field too wide (fault)
- Invalid combination of bits in PSL restored by REI (fault)
- Reserved pattern operator in EDITPC (abort)
- Incorrect source string length at completion of EDITPC (abort)
- Invalid combination of bits in PSW/MASK longword during RET (fault)
- Invalid combination of bits in BISPSW/BICPSW (fault)
- Invalid CALLx entry mask (fault)
- Invalid register number in MFPR or MTPR (fault)
- Invalid combinations in PCB loaded by LDPCTX (abort)
- Unaligned operand in ADAWI, INSQU, or REMQUE (fault)
- Invalid register contents in some MTPRs (fault)

3.5.3.4 Exceptions as the Result of an Instruction (Faults or Traps) The following exceptions are detected during opcode evaluation or as the result of instruction execution.

Reserved Opcode Fault (DIGITAL) - This fault occurs when the CPU encounters an opcode that is reserved to Digital Equipment Corporation, is not specifically defined, or requires higher privileges than the current mode allows No additional parameters are pushed on the stack. The opcode FFFF (hex), for example, always causes this fault.

Reserved Opcode Fault (Customer or DIGITAL Computer Special Systems) - This fault occurs if an opcode reserved to customers or to the DIGITAL computer special systems (CSS) group in the xxFC (hex) range is executed The operation is the same as the opcode reserved to Digital Equipment Corporation fault except that the event is caused by a different set of opcodes and faults through a different vector.

Compatibility Mode Fault/Abort – The fault occurs when a reserved opcode or illegal instruction is encountered in compatibility mode. The abort may occur if an odd address error is detected during one of the following types of memory references:

- An instruction fetch
- Any reference with VA ⟨0⟩ equal to a 1 if the instruction is not a byte reference
- An address fetch in the evaluation of addressing mode 3, 5, or 7
- An index word fetch in the evaluation of addressing modes 6 and 7

An additional longword of information (trap code) is pushed onto the stack indicating which event caused the exception. The condition, trap code, and class of exception are shown in Table 3-64. Special opcodes and illegal instructions are detected by microbranches. Odd address errors are detected by a microtrap.

Table 3-64 Compatibility Mode Exceptions

Exception Condition	Trap Code	Class
Reserved opcode	0	Fault
BPT opcode	1	Fault
IOT opcode	2	Fault
EMT opcode	3	Fault
TRAP opcode	4	Fault
Illegal instruction	5	Fault
Odd address error	6	Abort

Breakpoint Fault This exception occurs when the breakpoint instruction (BPT) is executed. No parameters are pushed. To proceed from a breakpoint fault, a debugger or tracing program typically restores the original contents of the location containing the BPT, sets T in the PSL saved by the BPT fault, and then continues. When the breakpointed instruction is completed, a trace trap occurs. At this point, the tracing program can again reinsert the BPT instruction, restore T to its original state, and continue.

Trace Trap – This trap occurs between instructions when the trace (T) bit is set. Trace (T) is used for tracing programs for performance evaluation or for debugging purposes. It is designed so that only one trace trap occurs before execution of the next instruction. (A service routine invoked by CHMx and terminated by REI is considered to be a single instruction.) The PC saved on a trace trap is the address of the next instruction that would normally be executed.

To insure that one trace occurs per instruction in spite of other traps and faults, the PSL makes use of two bits, the trace enable (T) bit and trace pending (TP) bit. If only one bit were used, an interrupt occurring at the end of the instruction would either produce zero or two traces, depending on the design. Instead, the T bit is defined to produce a trap following any other traps or aborts. The trap is implemented by copying T to the TP bit which is then used to generate the exception. TP generates a fault at the start of the next instruction before any other processing takes place. The operating rules for tracing are as follows:

- TP is set at the beginning of an instruction if T is set.
- If the instruction faults or an interrupt is serviced, the pushed TP is cleared and the pushed PC is set to the start of the faulting or interrupt instruction.
- If the instruction aborts or takes an arithmetic trap, the pushed TP is set or cleared as the result of step 1.
- If an interrupt is serviced after instruction completion and arithmetic traps, but before tracing is checked at the start of the next instruction, then the pushed TP is set or cleared as the result of step 1.
- If TP is set at the beginning of an instruction, a trace pending fault takes place.

There are two special cases; the CHMx and REI instructions, the only instructions that may change TP. However, they also follow the above rules.

The routine entered by a CHMx is not traced because CHMx clears T and TP in the new PSL. However, if T was set at the beginning of CHMx, the saved PSL will have both T and TP set. REI traps if either T is set or if TP is set in the saved PSL. Because of this, the instruction sequence of CHMx through REI acts as a single instruction. The trace trap that occurs after an REI that has TP set before being executed, always occurs with the new PSL. Special care must therefore be taken if exception or interrupt routines are to be traced.

The CALLx instructions save a cleared T, although T in the PSL is unchanged. This prevents a debugger or trace program that proceeds from a BPT fault from getting a spurious trace from the RET that matches the CALL.

Interrupts and other exception detections occur before the detection of a trace trap. However, this causes no difficulties because the entire PSL (including T and TP) is saved on the initiation of an interrupt or exception and is restored at the end with an REI. Thus, interrupts and benign exceptions are transparent to the executing program.

Change Mode Trap - When execution of any of the change mode instructions (CHMK, CHME, CHMS, CHMU) completes, a trap is performed. Additional parameters pushed on the stack include the sign extended operand held by the D register. (This trap condition is detected by a microbranch.)

3.5.3.5 Arithmetic Traps – Arithmetic traps occur as the result of arithmetic or conversion operations, indicating that the exception occurred during the last instruction and that the instruction has completed. The traps are mutually exclusive and are assigned the same vector of 34 (hex).

Arithmetic trap conditions are detected by microbranches. The PC pushed on the stack is the address of the next instruction to be executed. In addition to the PSL and PC, a longword is also pushed on the stack that identifies the exception condition. The trap codes pushed on the stack for these exception conditions are shown in Table 3-65.

Table 3-65 Arithmetic Trap Conditions

Exception Condition	Trap Code
Integer overflow	1
Integer divide by zero	2
Floating overflow	3
Floating/decimal divide by zero	4
Floating underflow	5
Decimal overflow	6
Subscript range	7

The following paragraphs provide a brief description of the arithmetic trap conditions.

Integer Overflow Trap – This exception indicates that the last instruction executed had an integer overflow that set the V condition code bit, indicating that the integer overflow (IV) bit was set. The stored result is the low-order part of the correct result. N and Z are set according to the stored result and the type code pushed on the stack is a 1. The instructions RET, REI, REMQUE, MOVTUC, and BISPSW do not cause an overflow even if they set V. The EMODx, CVTFx, and CVTDx floating-point instructions may also cause an integer overflow.

Integer Divide-by-Zero Trap – This exception indicates that the last instruction executed had an integer zero divisor. The stored result is equal to the dividend and the V condition code bit is set. The type code pushed on the stack is 2.

Floating Overflow Trap – This exception indicates that the last instruction executed resulted in an exponent greater than 127 (unbiased) after normalization and rounding. The stored result contains a one in the sign and zeros in the exponent and fraction fields. This is a reserved operand and causes a reserved operand fault if used in a subsequent floating-point instruction. The N and V condition code bits are set and Z and C are cleared. The type code pushed on the stack is 3.

Floating/Decimal Divide-by-Zero Traps – The floating divide-by-zero trap indicates that the last instruction executed had a floating zero divisor. The stored result is the reserved operand (as described for the floating overflow trap) and the condition codes are set as for a floating overflow.

A decimal string divide by zero trap indicates that the last instruction executed had a decimal string zero divisor. The destination and condition codes are **unpredictable** and the zero divisor can be either +0 or –0. The type code pushed on the stack for both types of divide by zero exceptions is 4.

Floating Underflow Trap – This exception indicates that the last instruction executed resulted in an exponent less than −127 (unbiased) after normalization and rounding, and that the floating underflow (FU) bit was set. The stored result is zero. Except for POLYx, the N, V, and C condition codes are cleared and Z is set. In POLYx, the trap occurs on completion of the instruction, which may be many operations after the underflow. The condition codes are set on the final result in POLYx and the type code pushed on the stack is 5.

Decimal Overflow Trap – This exception indicates the last instruction executed had a decimal string result too large for the destination string provided and that the decimal overflow (DV) bit was set. The V condition code is always set and the type code pushed on the stack is 6.

Subscript Range Trap – This exception indicates that the last instruction was an INDEX instruction with a subscript operation that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand and the result is stored. The condition codes are set as if the subscript was within range and the type code pushed on the stack is 7.

3.5.3.6 Microtraps – Detection of certain conditions by the hardware causes the microprogram to trap to a service flow. Initiation of the trap causes the micro program counter (UPC) to be pushed on the microstack so that the microprogram can continue after the condition is serviced.

Because multiple microtrap conditions may occur at the same time, they are input to priority encoders as shown in Figure 3-86. If any of the conditions are present, the UTRAP signal is generated, asserting a UBEN field value of 10 (hex) from the branch logic. This value (UBEN = 10) selects microtrap vector bits UTRAP VECT ⟨03:00⟩ to form the lower four bits of the next microaddress (described in Sections 3.2.2.2 and 3.5.3). The remaining hardwired address bits, with the lower four bits, form the microtrap vector that points to the microcode service routine that handles the condition.

Microtrap conditions are serviced in the following priority order shown in Table 3-66.

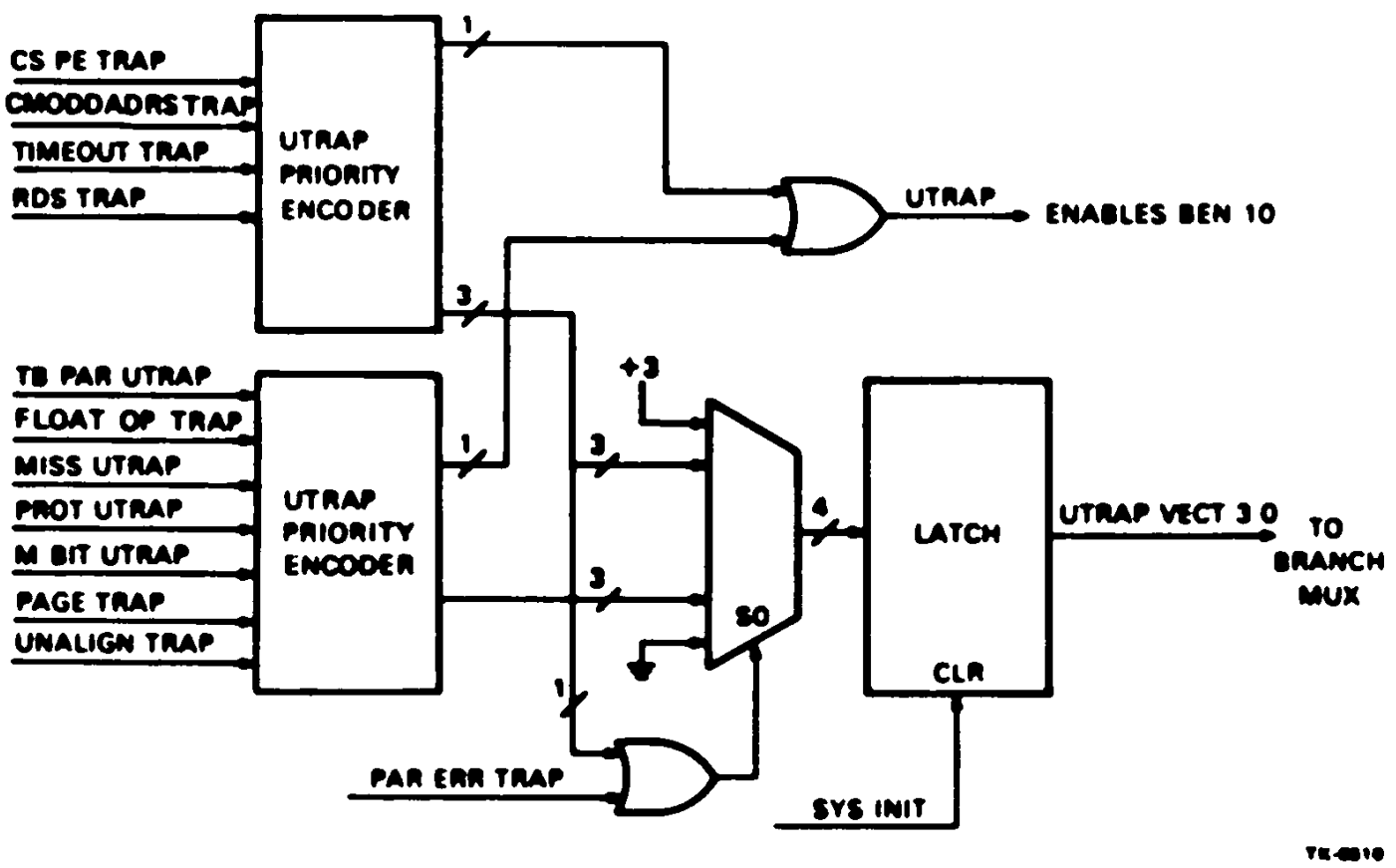


Figure 3-86 Microtrap Logic

Table 3-66 Microtrap Service Priorities

Microtrap Condition	Vector Assignment	Priority
System init	F00	Highest
CS parity error	E0F	↑
Odd address error	E0E	
Read timeout	E0D	
Read data substitute	E0C	
Cache parity error	E08	
TB parity error	E07	
Reserved floating operand	E06	
TB miss	E05	
Protection violation	E04	
Modify (M) bit	E03	
Page boundary	E02	
Unaligned data	E01	Lowest

Microtraps shown in Table 3-66 are initiated on the following conditions

- *System INIT* - Occurs when DC LO is asserted in the processor or DEAD is received from the SBI

- *CS Parity Error* - Occurs when the microsequencer detects a parity error in the next microinstruction May cause a machine check abort at any time

- *ODD Address Error* - Occurs on a 16-bit reference to an odd byte boundary in compatibility mode Causes an abort

- *Read Timeout* - Occurs under the following two conditions

  - The bus control logic could not gain access to the SBI or the addressed location responded with BUSY for 512 cycles

  - The bus control logic received a NO-RESPONSE confirmation, indicating a nonexistent address for 512 cycles.

- *Read Data Substitute* - Occurs when the processor performs a read or read-interlock on the SBI and memory returns uncorrected read data

- *Cache Parity Error* - Occurs when a parity error is detected in cache The address matrix and data matrix outputs are parity checked on each cache reference

- *TB Parity Error* - Occurs when a parity error is detected in the TB The information from both address and data matrix groups is parity checked when an address is asserted to the TB matrices

- *Reserved Floating Operand* Occurs when the microword UMSC field equals 2 (CHK FLOAT OP), ALU bit (15) is equal to 1, and ALU bits (14 07) are zeros

- *TB Miss* - Occurs when a valid requested page table entry is not found in the TB The PTE is fetched from main memory and placed in the TB during the microtrap service routine

- *Protection Violation* - Occurs if the current processor mode or intended page access violates the assigned protection for the page as directed by the PTE protection code

- *M Bit* - Occurs when a write is attempted to a page whose PTE contains a cleared M bit The PTE M bit is set in the TB and in memory during the microtrap service routine The PTE in memory is also fetched, modified, and rewritten

- *Page Boundary* Occurs when a cycle crossing a page boundary is attempted The intended access to the new page is checked during the microtrap service routine before the cycle can be executed This prevents writing the first part of a data stream where the writing of the second part would be prohibited

- *Unaligned Data* - Occurs when a reference is made across a longword boundary The microcode retrieves the portion of the data that was not part of the original longword in the microtrap service routine

3.5.4 Microword Control of Interrupts and Exceptions

The UIEK and UMSC fields of the microword control interrupts and exceptions as described in the following paragraphs

3.5.4.1 Interrupt and Exception Control (UIEK) Field The UIEK field of the microword (BUS CS (31 30)) is used to monitor and acknowledge interrupt conditions The UIEK field value specifies the functions shown in Table 3-67

Table 3-67 UIEK Field Interrupt and Exception Functions

UIEK Field Value	Function
0	NOP
1	Interrupt strobe (ISTR)
2	Interrupt acknowledge (IACK)
3	Exception acknowledge (EACK)

The interrupt strobe (ISTR) function clocks the hardware interrupt register (HIR), sampling the interrupt lines on levels (1E 14) and detecting interrupts in priority order The interrupt strobe is normally asserted at the end of an instruction before returning to the instruction decode (IRD) state If the priority of the interrupt is higher than the current IPL field in the PSL, an interrupt branch is performed at A fork in the microcode flow During the execution of long iterative instructions, the ISTR function is used to periodically monitor interrupt conditions and allow a subsequent branch to be performed

The interrupt acknowledge (IACK) function is used to clear the power-fail and console terminal receive and transmit interrupts as they are serviced by the microcode service routine

The exception acknowledge (EACK) function is used to clear pending arithmetic traps after they have been serviced or when other exceptions occur

The IACK and EACK functions set the PSL to the predetermined states shown in Table 3-68

3.5.4.2 **Miscellaneous (UMSC) Field** The UMSC field of the microword (BUS CS (29 26)) provides a number of functions, some of which affect the generation of exception conditions The function specified by each UMSC field value is shown in Table 3-69

Table 3-68 IACK and EACK Functions of the UIEK Field

PSL Bit	Name	IACK Function	EACK Function
(31)	CMP	0	0
(30)	TP	0	0
(29 28)	0	0	0
(27)	FPD	0	0
(26)	IS	IS	IS
(25 24)	CUR MOD	0	0
(23 22)	PREV MODE	0	CUR MODE
(21)	0	0	0
(20 16)	IPL	IPL ACT	IPL
(15 08)	0	0	0
(07)	DV	0	0
(06)	FU	0	0
(05)	IV	0	0
(04)	T	0	0
(03)	N	0	0
(02)	Z	0	0
(01)	V	0	0
(00)	C	0	0

Table 3-69 L MSC Field Miscellaneous Functions

L MSC Field Value	Function	Description
0	NOP	No operation
1	CHK CHMX INSTR	Loads the CUR MODE bits into the PREV MODE field of the PSL. If the new dataspecified by the change mode instruction is greater than the CUR MODE value, this function prevents loading CUR MODE
2	CHK FLOAT OP	Causes a microtrap before execution of the next microinstruction if ALU bit (15) is a 1 AND ALU bits (14 07) are 0
3	CHK ODD ADDRS	Causes a microtrap before executing the next microinstruction when in COMP MODE, VA (00) is a 1, and enabled by a memory cycle
4	Unused	
5	LOAD STATE	Loads contents of the state register to EAMX and enables the load function of the state register
6	LOAD ACC COND CODES	Loads the PSL condition codes from the ACC condition codes clocked on the previous microinstruction
7	READ RLOG	Loads the RLOG and PCSV inputs into the BMX register and decrements the RLOG pointer at the end of the microinstruction
8	IRD STATE	Indicates that a new macroinstruction has begun to be evaluated. Causes the RLOG pointer to be initialized, TP to be loaded from PSL T, and HP to be loaded from the halt request signal
9	Unused	
A	CLR NESTED ERROR	Clears the nested error flag in the CPU error/status register
B	SET NESTED ERROR	Sets the nested error flag in the CPU error/status register
F	INH COMP MODE	Prevents the PSL CMP bit from forcing VAMX (31 16) to zeros. Also inhibits the odd address microtrap

Table 3-69 UMSC Field Miscellaneous Functions (Cont)

UMSC Field Value	Function	Description
The following functions are used with the UMCT and UADS fields of the microword to further specify a memory reference		
C	SEC REF, INH TRAPS, SAVED CTX	
D	INH TRAPS, SAVED CTX	
E	SAVED CTX	
		SEC REF - Used to generate the secondpart of a byte or load mask when referencing unaligned data INH TRAPS - Prevents a page and unaligned microtrap from occurring SAVED CTX - Specifies the data type information saved by a previously specified UMCT code used to generate the byte and load masks Also used to detect odd address, page boundary, and unaligned data microtraps

3.6 SYSTEM CLOCKS

The VAX-11/785 contains three clocks the processor clock, time-of-day clock, and interval timer Each performs the functions described in this section

3.6.1 Processor Clock

The M7474 clock (CLK) module shown in Figure 3-87 is in slot 16 of the CPU backplane The CLK module performs the following functions

- Generates and synchronizes the CPU and SBI timing pulses.
- Decodes the CPU time states (CPT 0, CPT 1, CPT 2, CPT 3) and distributes them to modules in the CPU backplane
- Sends the SBI timing pulses to the TRS module where they are decoded to the SBI time states (SBI T0, SBI T1, SBI T2, SBI T3) From the TRS module, they are asserted to the SBH and SBL modules and the SBI
- SBI time states are also decoded on the CLK module to drive the four LED indicators on the module handle The decoded time state SBI T2 is also sent to the CIB module
- Clock Phase Error (CPE) - Phase detection logic monitors the CPU and SBI time and phase relationships and asserts the CPE bit to the CIB and TRS modules
- ACLO/DCLO - The CLK module monitors the ACLO and DCLO signals from the system power supplies and produces the CPU power-up and power-fail sequences

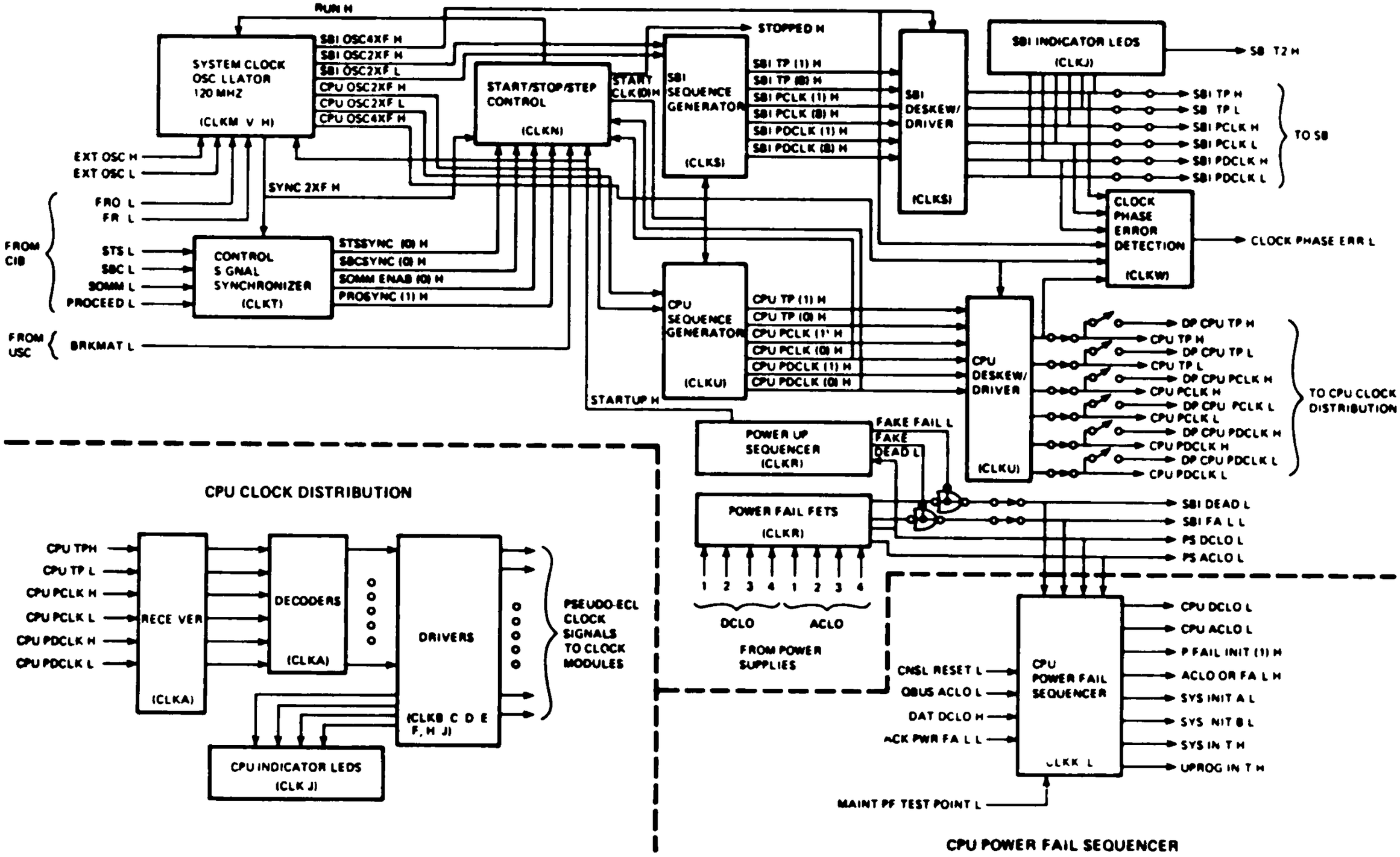


Figure 3-87 Clock (CLK) Module Block Diagram

Because the high-speed ECL logic is static sensitive, recommended test procedures must be carefully followed The logic is also sensitive to capacitive coupling and careful probing must be performed to obtain accurate measurements Short ground leads are recommended and bayonet probes must be used if critical timing measurements are to be made

Recommended test equipment includes a sensitive oscilloscope such as the Tektronix 7904™ (500 picosecond time base) with P6201 high-speed probes (1 5 picofarad)

3.6.1.1 Clock Phase and Error Detection - The system clock oscillator frequency output is divided by four to generate the CPU time base It is divided by six to generate the SBI time base, producing a three-to-two timing ratio between them

Synchronous CPU operation is based on a 133-nanosecond cycle that consists of four 33-nanosecond time states A 200-nanosecond SBI cycle consists of four 50-nanosecond time states On the detection of an out-of-phase condition, the clock phase error (CPE) signal is asserted to the CIB module where it is read by the console as a machine control/status register bit (MCS (15)). It is also sent to the TRS module where it asserts the SBI TR00 signal (HOLD) line on the SBI to temporarily prevent arbitration

3.6.1.2 Frequency Selection - The system clock oscillator frequency is selected from one of three internal oscillators or an external source by two fequency select bits (FREQ (01 00)) on the CIB module The console selects these bits as machine control register (MCR) bits during maintenance to control system timing as shown in Table 3-70

Table 3-70 MCR Control of System Clock Frequency

MCR Bits (04)	MCR Bits (03)	System Clock Oscillator Frequency	CPU Cycle Time	SBI Cycle Time
0	0	120 Mhz (normal)	133 333 ns	200 0 ns
0	1	123 Mhz (2.5% fast)	130 000 ns	205 0 ns
1	0	108 Mhz (10% slow)	148 148 ns	180.0 ns
1	1	External source		

A CPU cycle is from CPT0 to CPT0 (rising edge to rising edge) A SBI cycle is from SBI T0 to SBI T0 (rising edge to rising edge)

3.6.1.3 Start/Step/Step Control - On a CPU power-up sequence, clocks are held off by the power-up sequencer until the system clock oscillator is stable. When STARTUP is asserted, synchronized clocks are turned on and started in lockstep They are then released to the system while system initialize (INIT) is still asserted

The console has access to four machine control/status register bits on the CIB module that control the system clock in maintenance mode The bits are asserted to the CLK module where they are latched by the control signal synchronizer and asserted to the start/stop/step control logic The four bits control CPU clock operation as shown in Table 3-71

Tektronix 7904™ is a trademark of Tektronix, Inc

Table 3-71 MCR Control of CPU Clock Operation

MCR Bit	Function
⟨06⟩	*Stop on Microbreak Match (SOMM)* - The console stops the CPU clock at a specific microaddress by writing the microaddress to the microprogram break (MBRK) register. With SOMM set, the CPU clock then stops on the second CPT 0 following the CPU cycle in which a match occurs A match occurs when the MBRK register contents are equal to the microprogram counter (UPC) contents, asserting BRKMAT to the start/stop/step control from the microsequencer (USC) module
⟨02⟩	*Single Time State (STS)* - When first set, the STS bit stops the CPU clock in one of the four time states. The console then steps the CPU clock by one time state by writing a 1 to the PROCEED bit. The console is able to determine the CPU and SBI clock phase sequence by stepping the CPU clock and reading the state of SBI T2 from MCR bit ⟨13⟩.
⟨01⟩	*Single Bus Cycle (SBC)* - When SBC is set (and STS is clear), the CPU clock stops on the next CPT 0. The console then steps the CPU clock by one CLK cycle (to the next CPT 0) by writing a 1 to the PROCEED bit.
⟨00⟩	*PROCEED* - Writing a 1 to the PROCEED bit affects the CPU clock in one of four ways depending on the following states of the STS and SBC bits The CPU clock and SBI clock remain synchronized through all operations. Writing a 1 to the PROCEED bit while the clock is running has no effect. STS SBC Effect on CPU Clock 0 0 Sets clock 0 1 Steps clock one cycle 1 0 Steps clock one time state 1 1 (Unpredictable results)

3.6.1.4 Processor Timing – The CPU and SBI sequence generators output three sets of phase-related timing signals Each signal name begins with a prefix that defines its function, timing pulse (TP), phase clock (PCLK), or phase delayed clock (PDCLK)

The CPU clock distribution logic receives and decodes the CPU timing signals, sending CPU time states CPT 0 through CPT 3 to modules in the CPU backplane. The SBI timing signals are sent to the SBI terminator and SILO (TRS) module where they are decoded to SBI time states SBI T0 through SBI T3 and asserted on the SBI Decoding on the CLK module is used to drive the SBI time state indicators and decoded SBI T2 is sent to the CIB module where it is read as control register bit MCR ⟨13⟩ by the console program.

Figure 3-88 shows the phase relationship of the CPU and SBI time states pulses. They are in lockstep when CPT0 is in phase with SBI T2. (SBI T0 is in phase with CPT1 or CPT3 on every other SBI cycle.) SBI T2 is also sent to the CIB module. During maintenance, the console program monitors SBI T2 along with the CPU time states to determine when the two time bases are in lockstep.

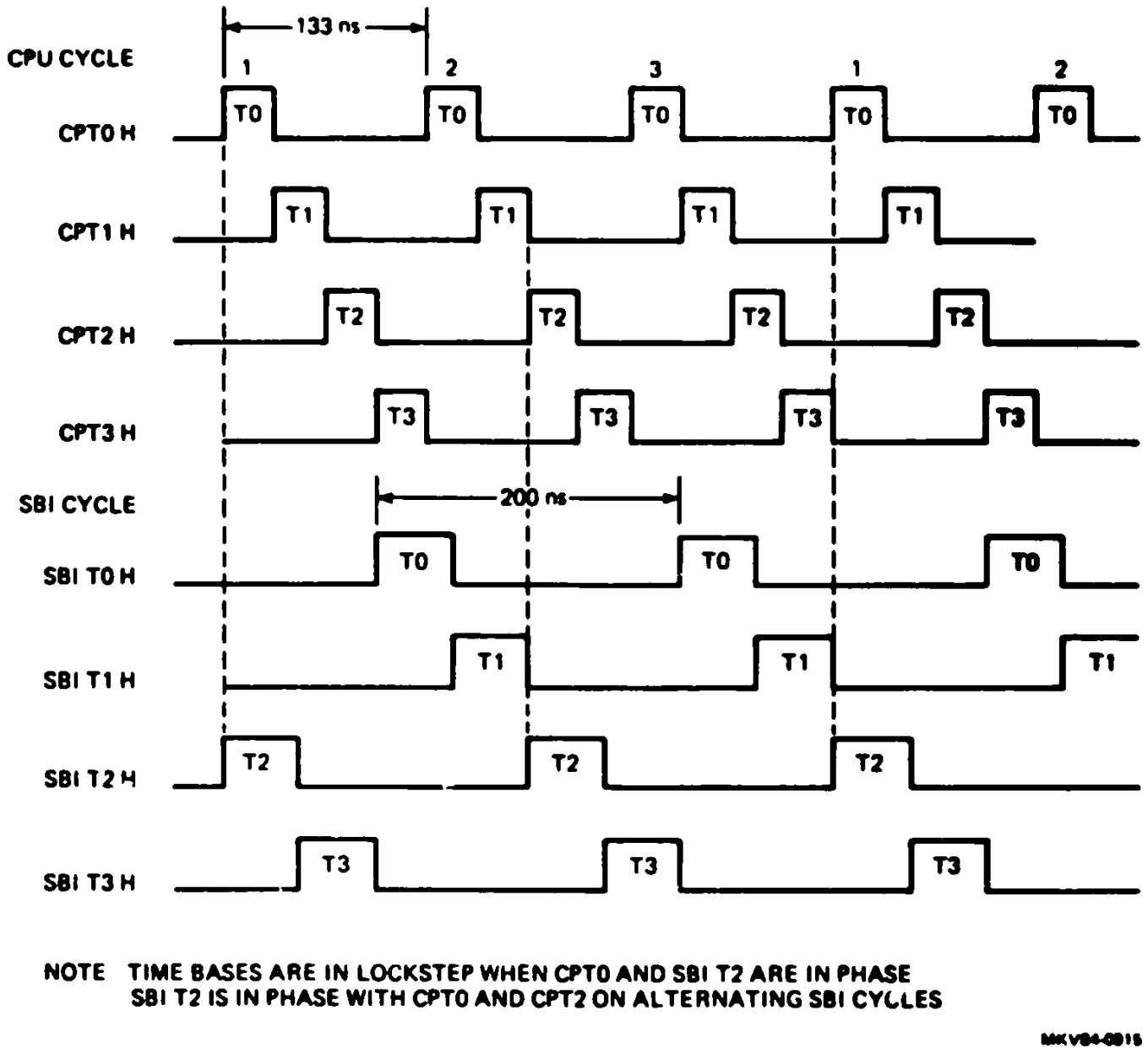


Figure 3-88 CPU and SBI Time States

Figure 3-89 shows how the CPU time states are decoded from the three sets of CPU timing signals by the CPU clock distribution logic on the CLK module. Figure 3-90 shows how the SBI time states are decoded on the TRS module from the three sets of SBI timing signals. All single cycle or single step functions are determined with respect to the CPU clock.

When the CPU clock is single stepped by the console, the SBI time states follow the protocol shown in Figure 3-88, either remaining the same or changing to the next state. There are three CPU cycles from lockstep to lockstep. Stopping in lockstep is assured when the CPU clock is stopped by setting SBC (MCR ⟨01⟩). With STS (MCR ⟨02⟩) set, stepping through successive time states using the PROCEED bit steps the clock to the next time state regardless of the state of MCR ⟨01⟩.

Figure 3-91 shows the action of the power-up sequencer when system power is applied. Figure 3-92 shows the action of the power fail sequencer when system power is removed. ACLO and DCLO from the system power supplies initiate the power-up and power-fail sequences as shown in Figure 3-87.

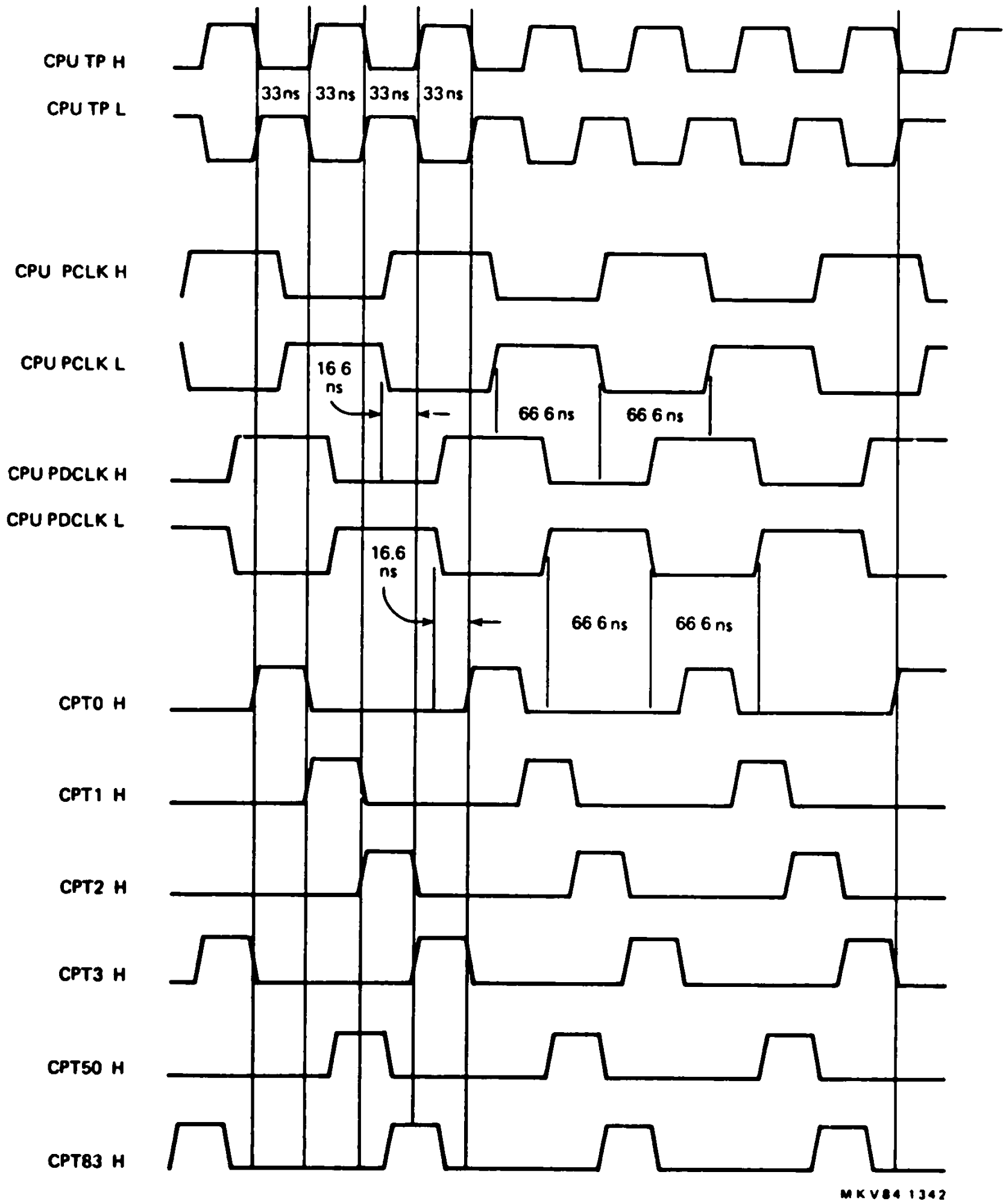


Figure 3-89 CPU Time State Generator Timing

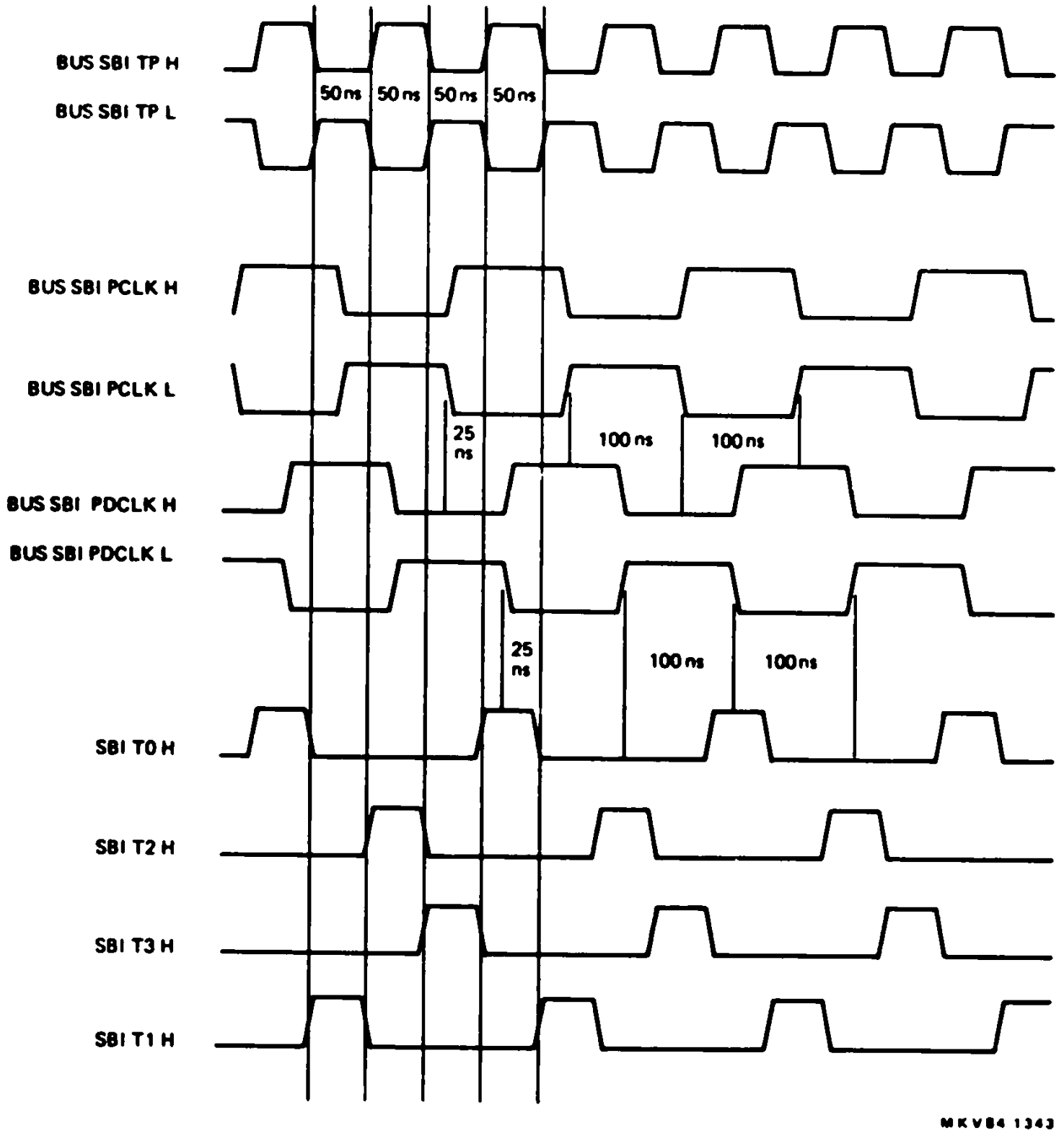


Figure 3-90 SBI Time State Generator Timing

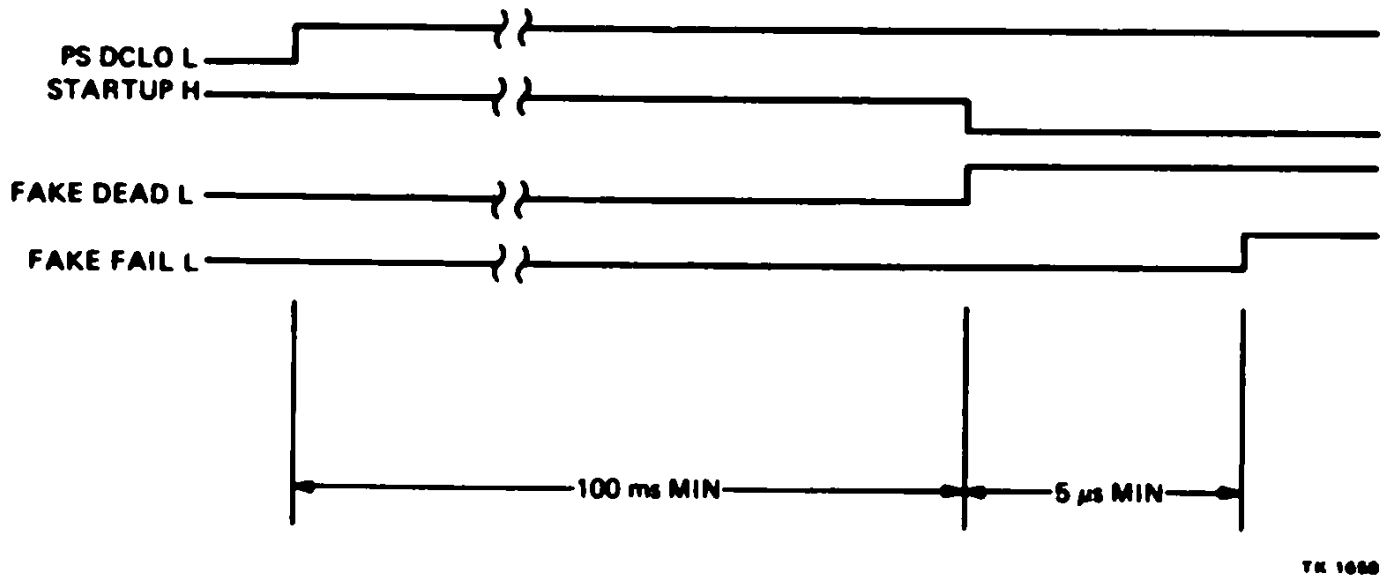
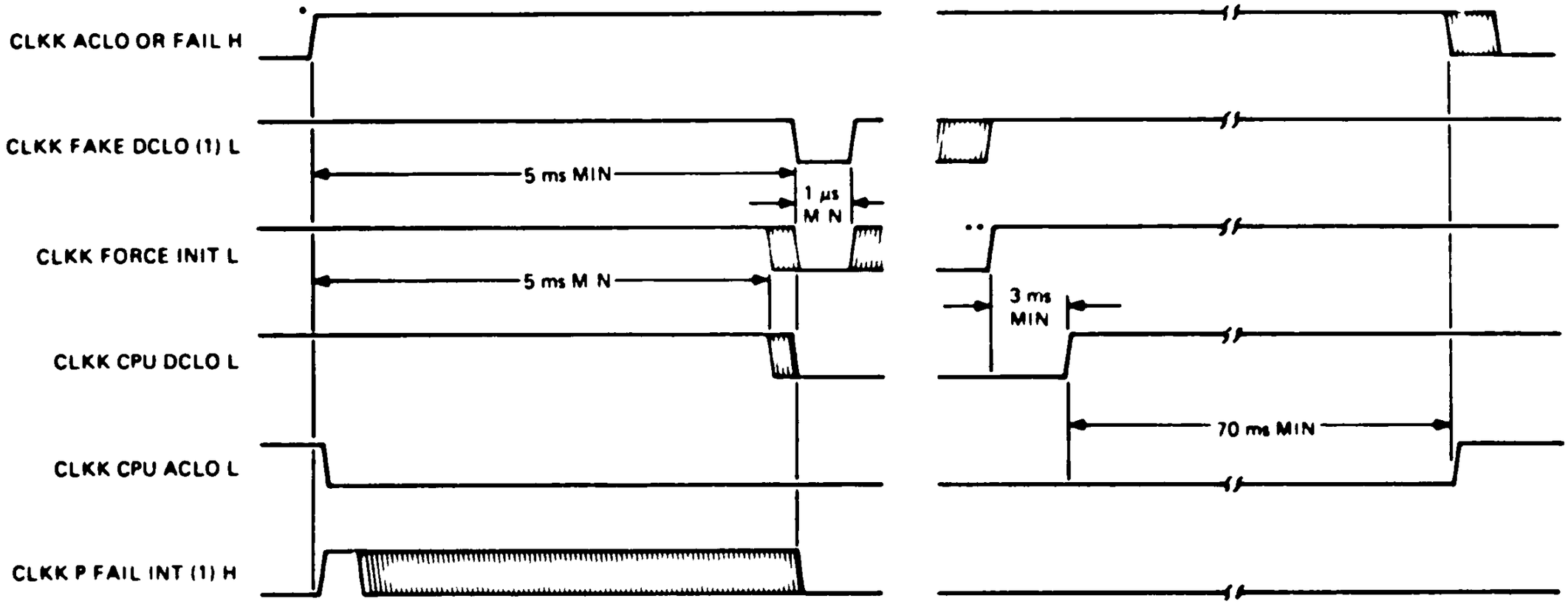


Figure 3-91 CPU Power Up Sequencer Timing



POWER DOWN SEQUENCE

*TRIGGERED FROM THE FIRST ASSERTION OF AN ACLO EQUIVALENT SIGNAL

POWER-UP SEQUENCE

**TRIGGERED FROM THE LAST DEASSERTION OF A DCLO EQUIVALENT SIGNAL

ACLO EQUIVALENT SIGNALS	CLKR PS ACLO L BUS SBI FAIL L CIBV QBUS ACLO L IRCM BAT DCLO H CLKL MAINT PF (1) H
DCLO EQUIVALENT SIGNALS	CLKR PS DCLO L BUS SBI DEAD L CLKK FAKE DCLO (1) L

Figure 3-92 CPU Power Fail Sequencer Timing

3.6.2 Time-of-Day Clock (TODC)

The time-of-day clock is used by the software to perform timekeeping functions. Its main purpose is to provide the correct time to the system after a power failure, eliminating the need for an operator to enter the time on the system restart.

The time-of-day clock logic shown in Figure 3-93 is located on the M7466 instruction decode (IRC) module in slot 8 and is accessed by the software with the MTPR and MFPR instructions. The time of day register is a 32-bit binary up counter that counts at a 100-Hertz rate, providing a range of 497.1 days.

New data written to the register is first clocked to a holding register from the ID bus. It is then loaded to the time-of-day register when the 100-Hertz clock synchronizes on the ID write to the holding register.

On a read operation, a special microprogram flow is required because of the slow switching speed of CMOS. In order to synchronize the 100-Hertz counter to the CPU, the microprogram reads the register twice and compares the two values. If they are the same, the value is sent to the software. If they are different, the microprogram continues to test until the values are the same.

The software converts the time input by the operator to a binary number that represents the year, month, day, hour, etc. When the software reads the TODC register, it then converts the current value to the correct form for output. (At the year's end, the software resets the clock to the beginning of the new year value.)

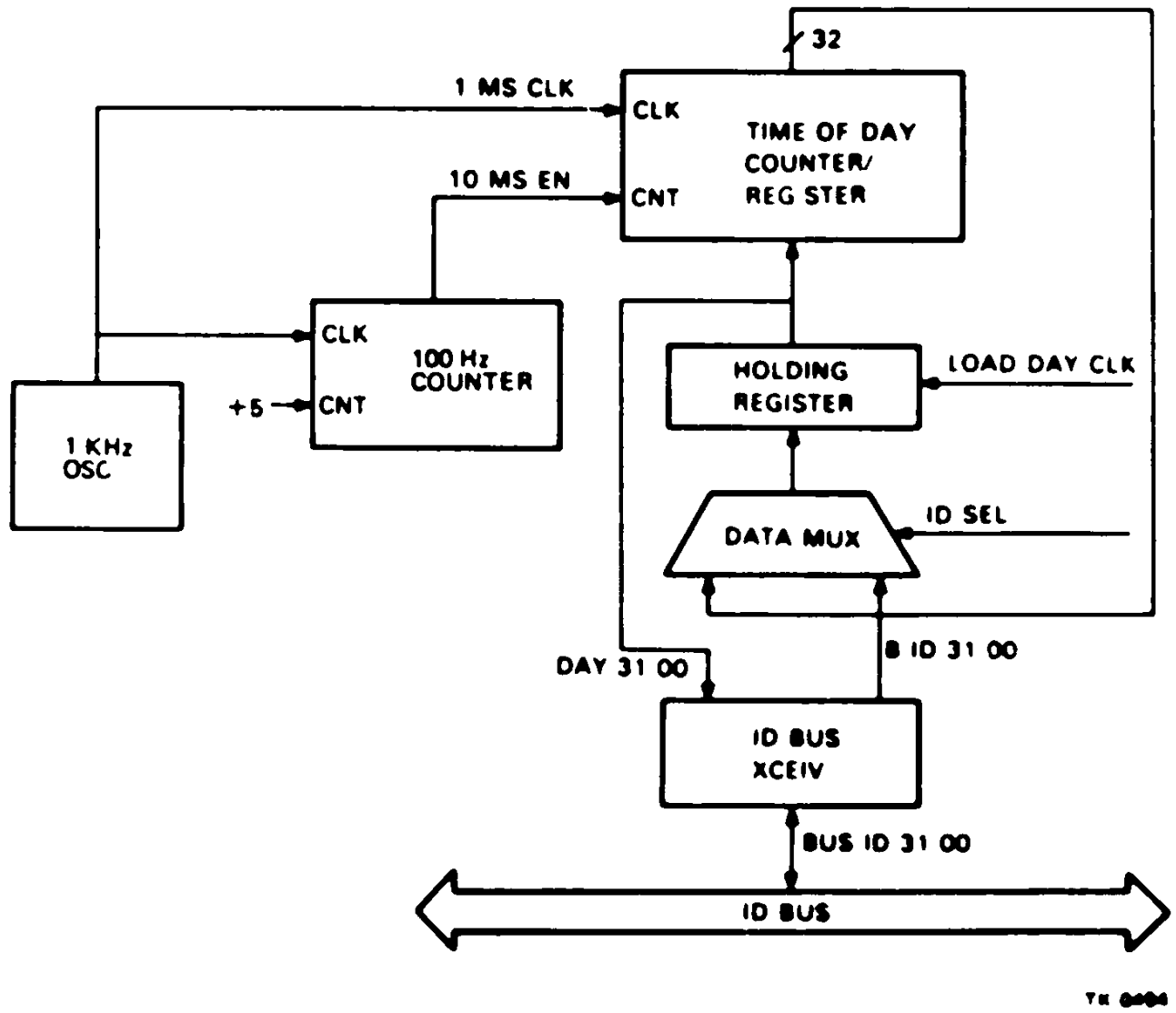


Figure 3-93 Time-of-Day Clock (TODC)

3.6.3 Interval Timer

The interval timer is located on the ICL module in slot 15 and operates at a 1 microsecond rate It allows the accurate measurement of real time intervals by interrupting the CPU on the completion of a specified time interval, allowing the software to perform time dependent events, accounting, or to maintain data and execution times

Three registers are used for interval timer operation The registers are accessed by the MTPR and MFPR instructions and have the following functions

- *Interval Count Register (ICR)* - The ICR is a 32-bit binary counter that increments at a 1 μs rate when the RUN control bit is set in the ICCS register This is a read-only register

- *Next Interval Count Register (NICR)* - The NICR is a 32-bit register that holds the value loaded to the ICR each time it overflows This is a write-only register.

- *Interval Clock Control Status Register (ICCS)* - ICCS register bit usage is shown in Figure 3-94. This register consists of six bits that have functions shown in Table 3-72.

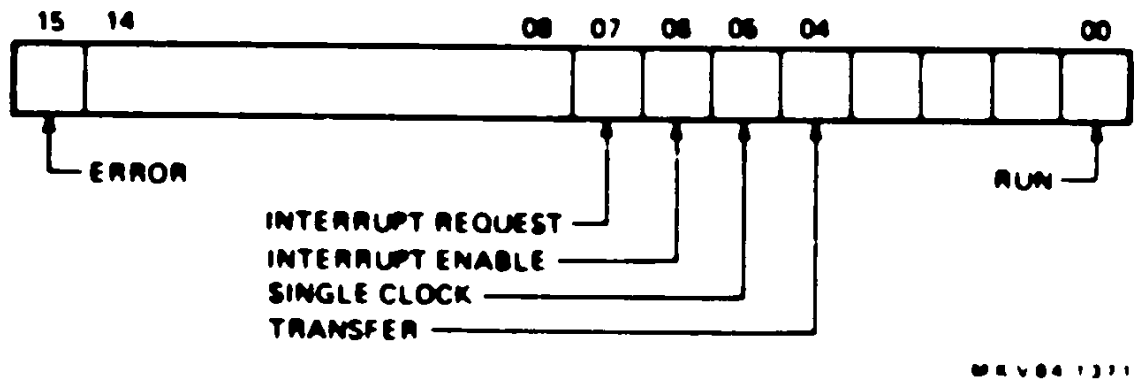


Figure 3-94 Interval Clock Control/Status (ICCS) Register

Table 3-72 KCCS Register Bit Functions

Bit	Name	Function
(15)	ERROR	This bit is set if a second overflow of the KCR occurs before the first overflow has been serviced. A read-only bit, it is cleared on a power up sequence or by writing a 1 to the KCCS(15) bit position.
(07)	INTERRUPT REQUEST	An KCR overflow first sets this bit, then clocks the contents of the NICR to the KCR. A read-only bit, it is cleared on a power up sequence or by writing a 1 to the KCCS(07) bit position.
(06)	INTERRUPT ENABLE	If this bit is set, an KCR overflow requests a CPU interrupt at IPL 24. A read/write bit, it is cleared on a power-up sequence.
(05)	SINGLE CLOCK	Writing a 1 to this bit increments the KCR by one for maintenance purposes. This is a write-only bit and always reads as zero.
(04)	TRANSFER	Writing a 1 to this bit, before starting the KCR, loads the contents of the NICR to the KCR (regardless of the state of the RUN bit). This is a write-only bit and always reads as zero.
(00)	RUN	When set, this bit enables the KCR for counting. A read/write bit, it is cleared on a power up sequence.

APPENDIX A

APPENDIX A
INSTRUCTION OPCODES

Opcode	Mnemonic	Instruction
00	HALT	Halt
01	NOP	No operation
02	REI	Return from exception or interrupt
03	BPT	Break point fault
04	RET	Return from called procedure
05	RSB	Return from subroutine
06	LDPCTX	Load process context
07	SVPCTX	Save process context
08	CVTPS	Convert packed to leading separate numeric
09	CVTSP	Convert leading separate numeric to packed
0A	INDEX	Compute index
0B	CRC	Calculate cyclic redundancy check
0C	PROBER	Probe read access
0D	PROBEW	Probe write access
0E	INSQUE	Insert into queue
0F	REMQUE	Remove from queue
10	BSBB	Branch to subroutine with byte displacement
11	BRB	Branch with byte displacement
12	BNEQ,	Branch on not equal,
	BNEQU	Branch on not equal unsigned
13	BEQL,	Branch on equal
	BEQLU	Branch on equal unsigned
14	BGTR	Branch on greater
15	BLEQ	Branch on less or equal
16	JSB	Jump to subroutine
17	JMP	Jump
18	BGEQ	Branch on greater or equal
19	BLSS	Branch on less
1A	BGTRU	Branch on greater unsigned
1B	BLEQU	Branch on less or equal unsigned
1C	BVC	Branch on overflow clear
1D	BVS	Branch on overflow set
1E	BGEQU	Branch on greater or equal unsigned,
	BCC	Branch on carry clear
1F	BLSSU	Branch on less unsigned,
	BCS	Branch on carry set

Opcode	Mnemonic	Instruction
20	ADDP4	Add packed 4 operand
21	ADDP6	Add packed 6 operand
22	SUBP4	Subtract packed 4 operand
23	SUBP6	Subtract packed 6 operand
24	CVTPT	Convert packed to trailing numeric
25	MULP	Multiply packed
26	CVTTP	Convert trailing numeric to packed
27	DIVP	Divide packed
28	MOVC3	Move character 3 operand
29	CMPC3	Compare character 3 operand
2A	SCANC	Scan for character
2B	SPANC	Span characters
2C	MOVC5	Move character 5 operand
2D	CMPC5	Compare character 5 operand
2F	MOVTC	Move translated characters
2F	MOVTUC	Move translated until character
30	BSBW	Branch to subroutine with word displacement
31	BRW	Branch with word displacement
32	CVTWL	Convert word to longword
33	CVTWB	Convert word to byte
34	MOVP	Move packed
35	CMPP3	Compare packed 3 operand
36	CVTPL	Convert packed to longword
37	CMPP4	Compare packed 4 operand
38	EDITPC	Edit packed to character
39	MATCHC	Match characters
3A	LOCC	Locate character
3B	SKPC	Skip character
3C	MOVZWL	Move zero-extended word to longword
3D	ACBW	Add compare and branch word
3F	MOVAW	Move address of word
3F	PUSHAW	Push address of word
40	ADDF2	Add F_floating 2 operand
41	ADDF3	Add F_floating 3 operand
42	SUBF2	Subtract F_floating 2 operand
43	SUBF3	Subtract F_floating 3 operand
44	MULF2	Multiply F_floating 2 operand
45	MULF3	Multiply F_floating 3 operand
46	DIVF2	Divide F_floating 2 operand
47	DIVF3	Divide F_floating 3 operand
48	CVTFB	Convert F_floating to byte
49	CVTFW	Convert F_floating to word
4A	CVTFL	Convert F_floating to longword
4B	CVTRFL	Convert rounded F_floating to longword
4C	CVTBF	Convert byte to F_floating

Opcode	Mnemonic	Instruction
4D	CVTWF	Convert word to F_floating
4E	CVTLF	Convert longword to F_floating
4F	ACBF	Add compare and branch F_floating
50	MOVF	Move F_floating
51	CMPF	Compare F_floating
52	MNEGF	Move negated F_floating
53	TSTF	Test F_floating
54	EMODF	Extended modulus F_floating
55	POLYF	Evaluate polynomial F_floating
56	CVTFD	Convert F_floating to D_floating
57		Reserved to Digital Equipment Corporation
58	ADAWI	Add aligned word interlocked
59		Reserved to Digital Equipment Corporation
5A		Reserved to Digital Equipment Corporation
5B		Reserved to Digital Equipment Corporation
5C	INSQHI	Insert into queue head, interlocked
5D	INSQTI	Insert into queue tail, interlocked
5E	REMQHI	Remove from queue head, interlocked
5F	REMQTI	Remove from queue tail interlocked
60	ADDD2	Add D_floating 2 operand
61	ADDD3	Add D_floating 3 operand
62	SUBD2	Subtract D_floating 2 operand
63	SUBD3	Subtract D_floating 3 operand
64	MULD2	Multiply D_floating 2 operand
65	MULD3	Multiply D_floating 3 operand
66	DIVD2	Divide D_floating 2 operand
67	DIVD3	Divide D_floating 3 operand
68	CVTDB	Convert D_floating to byte
69	CVTDW	Convert D_floating to word
6A	CVTDL	Convert D_floating to longword
6B	CVTRDL	Convert rounded D_floating to longword
6C	CVTBD	Convert byte to D_floating
6D	CVTWD	Convert word to D_floating
6E	CVTLD	Convert longword to D_floating
6F	ACBD	Add compare and branch D_floating
70	MOVD	Move D_floating
71	CMPD	Compare D_floating
72	MNEGD	Move negated D_floating
73	TSTD	Test D_floating
74	EMODD	Extended modulus D_floating
75	POLYD	Evaluate polynomial D_floating
76	CVTDF	Convert D_floating to F_floating
77		Reserved to Digital Equipment Corporation

Opcode	Mnemonic	Instruction
78	ASHL	Arithmetic shift longword
79	ASHQ	Arithmetic shift quadword
7A	EMUL	Extended multiply
7B	EDIV	Extended divide
7C	CLRQ.	Clear quadword
	CLRD.	Clear D_floating.
	CLRG	Clear G_floating
7D	MOVQ	Move quadword
7E	MOVAQ.	Move address of quadword
	MOVAD.	Move address of D_floating
	MOVAG	Move address of G_floating
7F	PUSHAQ.	Push address of quadword
	PUSHAD	Push address of D_floating
	PUSHAG	Push address of G_floating
80	ADDB2	Add byte 2 operand
81	ADDB3	Add byte 3 operand
82	SUBB2	Subtract byte 2 operand
83	SUBB3	Subtract byte 3 operand
84	MULB2	Multiply byte 2 operand
85	MULB3	Multiply byte 3 operand
86	DIVB2	Divide byte 2 operand
87	DIVB3	Divide byte 3 operand
88	BISB2	Bit set byte 2 operand
89	BISB3	Bit set byte 3 operand
8A	BICB2	Bit clear byte 2 operand
8B	BICB3	Bit clear byte 3 operand
8C	XORB2	Exclusive OR byte 2 operand
8D	XORB3	Exclusive OR byte 3 operand
8E	MNEGB	Move negated byte
8F	CASEB	Case byte
90	MOVB	Move byte
91	CMPB	Compare byte
92	MCOMB	Move complemented byte
93	BITB	Bit test byte
94	CLRB	Clear byte
95	TSTB	Test byte
96	INCB	Increment byte
97	DECB	Decrement byte
98	CVTBL	Convert byte to longword
99	CVTBW	Convert byte to word
9A	MOVZBL	Move zero-extended byte to longword
9B	MOVZBW	Move zero-extended byte to word
9C	ROTL	Rotate longword
9D	ACBB	Add compare and branch byte
9E	MOVAB	Move address of byte
9F	PUSHAB	Push address of byte

Opcode	Mnemonic	Instruction
A0	ADDW2	Add word 2 operand
A1	ADDW3	Add word 3 operand
A2	SUBW2	Subtract word 2 operand
A3	SUBW3	Subtract word 3 operand
A4	MULW2	Multiply word 2 operand
A5	MULW3	Multiply word 3 operand
A6	DIVW2	Divide word 2 operand
A7	DIVW3	Divide word 3 operand
A8	BISW2	Bit set word 2 operand
A9	BISW3	Bit set word 3 operand
AA	BICW2	Bit clear word 2 operand
AB	BICW3	Bit clear word 3 operand
AC	XORW2	Exclusive OR word 2 operand
AD	XORW3	Exclusive OR word 3 operand
AE	MNEGW	Move negated word
AF	CASEW	Case word
B0	MOVW	Move word
B1	CMPW	Compare word
B2	MCOMW	Move complemented word
B3	BITW	Bit test word
B4	CLRW	Clear word
B5	TSTW	Test word
B6	INCW	Increment word
B7	DECW	Decrement word
B8	BISPSW	Bit set processor status word
B9	BICPSW	Bit clear processor status word
BA	POPR	Pop register
BB	PUSHR	Push register
BC	CHMK	Change mode to kernel
BD	CHME	Change mode to executive
BE	CHMS	Change mode to supervisor
BF	CHMU	Change mode to user
C0	ADDL2	Add longword 2 operand
C1	ADDL3	Add longword 3 operand
C2	SUBL2	Subtract longword 2 operand
C3	SUBL3	Subtract longword 3 operand
C4	MULL2	Multiply longword 2 operand
C5	MULL3	Multiply longword 3 operand
C6	DIVL2	Divide longword 2 operand
C7	DIVL3	Divide longword 3 operand
C8	BISL2	Bit set longword 2 operand
C9	BISL3	Bit set longword 3 operand
CA	BICL2	Bit clear longword 2 operand
CB	BICL3	Bit clear longword 3 operand

Opcode	Mnemonic	Instruction
CC	XORL2	Exclusive OR longword 2 operand
CD	XORL3	Exclusive OR longword 3 operand
CE	MNEGL	Move negated longword
CF	CASEL	Case longword
D0	MOVL	Move longword
D1	CMPL	Compare longword
D2	MCOML	Move complemented longword
D3	BITL	Bit test longword
D4	CLRL	Clear longword
	CLRF	Clear F_floating
D5	TSTL	Test longword
D6	INCL	Increment longword
D7	DECL	Decrement longword
D8	ADWC	Add with carry
D9	SBWC	Subtract with carry
DA	MTPR	Move to processor register
DB	MFPR	Move from processor register
DC	MOVPSL	Move processor status longword
DD	PUSHL	Push longword
DE	MOVAL	Move address of longword
	MOVAF	Move address of F_floating
DF	PUSHAL	Push address of longword
	PUSHAF	Push address of F_floating
E0	BBS	Branch on bit set
E1	BBC	Branch on bit clear
E2	BBSS	Branch on bit set and set
E3	BBCS	Branch on bit clear and set
E4	BBSC	Branch on bit set and clear
E5	BBCC	Branch on bit clear and clear
E6	BBSSI	Branch on bit set and set, interlocked
E7	BBCCI	Branch on bit clear and clear, interlocked
E8	BLBS	Branch on low bit set
E9	BLBC	Branch on low bit clear
EA	FFS	Find first set bit
EB	FFC	Find first clear bit
EC	CMPV	Compare field
ED	CMPZV	Compare zero-extended field
EE	EXTV	Extract field
EF	EXTZV	Extract zero-extended field
F0	INSV	Insert field
F1	ACBL	Add compare and branch longword
F2	AOBLSS	Add one and branch on less
F3	AOBLEQ	Add one and branch on less or equal

Opcode	Mnemonic	Instruction
F4	SOBGEQ	Subtract one and branch on greater or equal
F5	SOBGTR	Subtract one and branch on greater
F6	CVTLB	Convert longword to byte
F7	CVTLW	Convert longword to word
F8	ASHP	Arithmetic shift and round packed
F9	CVTLP	Convert longword to packed
FA	CALLG	Call with general argument list
FB	CALLS	Call with stack argument list
FC	XFC	Extended function call
FD	ESCD to DEC	
FE	ESCE to DEC	
FF	ESCF to DEC	

APPENDIX B

APPENDIX B
EXTENDED-PRECISION OPCODES

Opcode	Mnemonic	Instruction
00FD to 31FD		Reserved to Digital Equipment Corporation
32FD	CVTDH	Convert D_floating to H_floating
33FD	CVTGF	Convert G_floating to F_floating
34FD to 3FFD		Reserved to Digital Equipment Corporation
40FD	ADDG2	Add G_floating 2 operand
41FD	ADDG3	Add G_floating 3 operand
42FD	SUBG2	Subtract G_floating 2 operand
43FD	SUBG3	Subtract G_floating 3 operand
44FD	MULG2	Multiply G_floating 2 operand
45FD	MULG3	Multiply G_floating 3 operand
46FD	DIVG2	Divide G_floating 2 operand
47FD	DIVG3	Divide G_floating 3 operand
48FD	CVTGB	Convert G_floating to byte
49FD	CVTGW	Convert G_floating to word
4AFD	CVTGL	Convert G_floating to longword
4BFD	CVTRGL	Convert rounded G_floating to longword
4CFD	CVTBG	Convert byte to G_floating
4DFD	CVTWG	Convert word to G_floating
4EFD	CVTLG	Convert longword to G_floating
4FFD	ACBG	Add, compare and branch G_floating
50FD	MOVG	Move G_floating
51FD	CMPG	Compare G_floating
52FD	MNEGG	Move negated G_floating
53FD	TSTG	Test G_floating
54FD	EMODG	Extended modulus G_floating
55FD	POLYG	Polynomial evaluation G_floating
56FD	CVTGH	Convert G_floating to H_floating
57FD		Reserved to Digital Equipment Corporation
58FD to 5FFD		Reserved to Digital Equipment Corporation

Opcode	Mnemonic	Instruction
60FD	ADDH2	Add H_floating 2 operand
61FD	ADDH3	Add H_floating 3 operand
62FD	SUBH2	Subtract H_floating 2 operand
63FD	SUBH3	Subtract H_floating 3 operand
64FD	MULH2	Multiply H_floating 2 operand
65FD	MULH3	Multiply H_floating 3 operand
66FD	DIVH2	Divide H_floating 2 operand
67FD	DIVH3	Divide H_floating 3 operand
68FD	CVTHB	Convert H_floating to byte
69FD	CVTHW	Convert H_floating to word
6AFD	CVTHL	Convert H_floating to longword
6BFD	CVTRHL	Convert rounded H_floating to longword
6CFD	CVTBH	Convert byte to H_floating
6DFD	CVTWH	Convert word to H_floating
6EFD	CVTLH	Convert longword to H_floating
6FFD	ACBH	Add, compare and branch H_floating
70FD	MOVH	Move H_floating
71FD	CMPH	Compare H_floating
72FD	MNEGH	Move negated H_floating
73FD	TSTH	Test H_floating
74FD	EMODH	Extended modulus H_floating
75FD	POLYH	Polynomial evaluation H_floating
76FD	CVTHG	Convert H_floating to G_floating
77FD		Reserved to Digital Equipment Corporation
78FD to 7BFD		Reserved to Digital Equipment Corporation
7CFD	CLRH, CLRO	Clear H_floating, Clear octaword
7DFD	MOVO	Move octaword
7EFD	MOVAH, MOVAO	Move address of H_floating, Move address of octaword
7FFD	PUSHAH, PUSHAO	Push address of H_floating, Push address of octaword
80FD to 97FD		Reserved to Digital Equipment Corporation
98FD	CVTFH	Convert F_floating to H_floating
99FD	CVTFG	Convert F_floating to G_floating
9AFD to FDFD		Reserved to Digital Equipment Corporation

Opcode	Mnemonic	Instruction
F6FD	CVTHF	Convert H_floating to F_floating
F7FD	CVTHD	Convert H_floating to D_floating
F8FD to FCFF		Reserved to Digital Equipment Corporation
FDFF	BUGL	Bugcheck longword
FEFF	BUGW	Bugcheck word
FFFF		Permanently reserved